2

RADC-TR-78-58
Interim Report
April 1978

A COMPARISON OF PROGRAMMING LANGUAGES FOR SOFTWARE ENGINEERING

Mary Shaw
Guy T. Almes
Joseph M. Newcomer
Brian K. Reid
William A. Wulf

Carnegie-Mellon University

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-78-58 has been reviewed and approved for publication.

APPROVED: Douglas White
DOUGLAS WHITE
Project Engineer
Software Sciences Section

APPROVED: Wendall C Bauman
WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS, Acting Chief
Plans Office

f1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 **REPORT DOCUMENTATION PAGE**	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-78-58 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE ~~(and Subtitle)~~ 6 A Comparison of Programming Languages for Software Engineering.	5. ~~TYPE OF REPORT~~ & PERIOD COVERED 9 Interim Report,
	6. PERFORMING ORG. REPORT NUMBER N/A
7. ~~AUTHOR(s)~~ 10 Mary Shaw, Guy T. Almes, Joseph M. Newcomer, Brian K. Reid, William A. Wulf	8. CONTRACT OR GRANT NUMBER(s) 15 F30602-75-C-0218
9. ~~PERFORMING ORGANIZATION NAME AND ADDRESS~~ Carnegie-Mellon University, Dept. of Computer Science, Schenley Park, Pittsburgh PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 63728F J.O. 55500814
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE 11 Apr[redacted]978
	13. ~~NUMBER OF PAGES~~ 98 12 98p.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same 16 5550 17 08	15. SECURITY CLASS. ~~(of this report)~~ UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same	
18. SUPPLEMENTARY NOTES RADC Project Engineer: Douglas White (ISIS)	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Fortran, Cobol, Jovial, DOD-1, programming languages, language comparison, software engineering	

--variously called 'DOD-1', 'Strawman', 'Woodman' 'Tinman' and 'Ironman')

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Four programming languages (Fortran, Cobol, Jovial, and the proposed DOD standard) are compared in the light of modern ideas of good software engineering practice. The comparison begins by identifying a core for each language that captures the essential properties of the intent of the language designers. These core languages then serve as a basis for the discussion of the language philosophies and the impact of the language on gross program organization and on the use of individual statements.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081 JOB

Chapter 1
Introduction

This report is a comparison of four programming languages in the light of modern ideas of good software engineering practice. We include three existing languages (Fortran, Cobol, Jovial) and the proposed DoD standard for programming embedded computer systems.[1]

At least some of the acrimony that surrounds the discussion of the relative merits of various programming languages arises from a failure to draw a proper distinction between *comparison* and *evaluation*. In a comparison, the analysis is directed at distinctions and similarities of the languages, and possibly at the relative merits of particular features under certain circumstances. An evaluation, on the other hand, is aimed at making a judgement, hopefully in terms of a set of clearly-defined objectives. We wish to emphasize that our primary goal in this report is a *comparison* of languages. Our comparison focusses on those dimensions that bear on modern notions of program structure and good software engineering practice.

In preparing this report we first addressed the problem of language comparison itself. After an examination of existing language comparisons, we felt that a new approach was needed. Thus, to replace the customary exhaustive feature-by-feature comparative evaluation, we have devised a "language core" technique; this technique is described in detail below. We believe it to be a substantive improvement in language-comparison methodology.

One of the reasons for evaluating programming languages is to help select one for use in a particular project or organization. This selection process often engenders bitter debate, as typically there are no clear goals or unified interests. The results frequently depend on the prior language experience of the evaluators rather than on any objective criteria.

[1] *This language is variously called DOD-1, "Strawman", "Woodenman", "Tinman", and "Ironman".*

We believe, however, that objectivity in the comparison process is possible, and we have based this report on that assumption: Rational and objective comparisons can be made. An evaluation combines comparison with judgments about particular objectives and methods; it must be accomplished with clear goals and proper information. This report can provide the language-dependent part of the information needed to make a rational evaluation in a given context.

1.1. Comparison Methodology

There have been a substantial number of "evaluations" of various programming languages [c.f. Goodenough 76, Higman 77, Nicholls 75]. Generally these comparisons or evaluations have been based on a detailed analysis of the *features* available in each of the languages under study. Because our goal is more specific, namely comparison only with respect to the use of the language in conjunction with modern notions of software engineering, we have chosen a different approach.

Our approach is based on three closely-related premises about programming languages:

1. The designers of each language had a strong image of how programs are, or ought to be, written -- and that they designed the language accordingly.

2. The designers' image is strongly and consistently reflected throughout the language. Moreover, once a language is defined, it tends to acquire an independent identity. Extensions and revisions tend to continue in the original style, thus remaining true to the original image of program construction.

3. This reflected image is a major factor in determining how the language will actually be used. That is, we believe that the language designer's philosophy of program construction dominates the language so strongly that it dictates (at least to a first approximation) a "mind set" which, in turn, strongly impacts how

programs will be structured.[2]

If our goal is the comparison of languages with respect to the way they support software engineering, a consequence of these premises is that we should compare the central images of the languages and the programming paradigms embodied by those designs. We propose to do just that, albeit in a somewhat indirect manner. Our comparison methodology has two steps:

1. Capturing the designer's image with a definition of the central *core* of each language.

2. Comparing these core languages.

The remainder of this section elaborates these two steps and the rationale for this approach.

1.1.1 Language Cores

As stated above, we believe that the *language designers'* image of the programming paradigm is a major factor in determining the structure of programs written in the language. It would be ideal to compare these images directly and possibly to discuss the extent to which each language really does support its image. In most cases, of course, we have little or no direct evidence of the designers' intent. In fact, we do not believe that this image was necessarily explicit. The notion of a "core language" is our attempt to express what we infer to have been their intent.

Since we believe that the designers' image of the programming paradigm is strongly and consistently reflected throughout the language -- both in the choice of features and in the way the features are composed into programs -- it is generally not necessary to examine the entire language in order to see this image. In other words, it is possible to isolate a subset of the language that captures the image; we call such a subset a *core* of the language.

[2] A programmer who learns one programming language or who predominantly uses one language may mentally program in that language and transliterate into another which he uses less frequently. In such cases his use of the second language will not conform to the argument above; however, we are primarily concerned with the primary impact of language and not with secondary effects.

Thus our language comparisons begin with the definition of each of the language cores. We shall compare (only) these cores since they were designed to capture the attributes we wish to compare.

In general, a core of a language will not be unique; for example, it may be possible to select any of several features to capture some single aspect of the image. Since small cores are easier to compare than large ones, a core will usually omit much of a language and, in principle, need not be complete.[3] Moreover, the selection of which aspects to include or exclude from the core of a particular language will be solely a property of that language; one is not entitled to expect corresponding features, even when they exist, to appear in the cores of two languages being compared. Such features may contribute to the unique flavor of a language in one case but not another.

1.1.2 Comparison of Languages

Although we realize that focussing the comparison on the cores of the languages in question has some obvious shortcomings for certain purposes, we also believe it has significant advantages over attempts at more comprehensive comparisons. These advantages are all, in one form or another, assertions that it is possible to do a more complete, intellectually tractable, and scientifically honest comparison of a small number of things than a large number. More precisely:

1. We assert that the design philosophy of a language pervades all programs written in that language. Indeed, we believe that this has a far more important impact on program construction than the presence or absence of specific features. Since, by design, the core language captures the language design philosophy, comparison of the cores captures the first-order effects in program composition.

2. We believe that comparison of (only) the cores avoids a whole raft of silly, finger-pointing arguments -- those arguments of the flavor " ...in MY language there is an *X* construct"... "Well, in MY language there

3 That is, it is not necessary for the core to be capable of expressing an arbitrary computation.

isn't an explicit *X*, but you can do with a *Y*...". Such discussions may be interesting to some language designers, but they shed little light on the real merits of either language. To avoid such silly discussions,

a. We should accept the premise that all languages are computationally complete; that is, all are adequate for describing all computations. One language will undoubtedly be more convenient than another for particular isolated examples; but the reverse will usually be true for other examples. Thus these isolated cases serve little purpose.

b. We should recognize that all languages tend to accrete features to compensate for their most glaring initial deficiencies. Genuinely important facilities will migrate into all languages in some form. The way in which these facilities are included, however, is important: in nearly all cases they enter the language in a style consistent with the original design philosophy.

c. We should recognize that the simple presence of features is not a good indication of the worth of a language. A large collection of *ad hoc* features, patched together without thought to uniformity or elegance, will not produce a usable language.

The core language approach responds directly to each of these points.

3. The core language is probably "solid" with respect to its semantics. Even though all the languages in question have strong portability goals, from practical experience we know that the less-exercised features of a language are likely to be implemented incompatibly. By staying within the center of the language we avoid disagreements about the semantics of strange cases.

4. Most importantly, we believe that the "core language" approach is intellectually manageable. Any "more complete" treatment would, of necessity, involve a lengthy enumeration of comparisons. It would

then to be difficult to determine (a) what the most important comparisons are and (b) whether crucial comparisons are missing.

1.2. Dimensions of Comparison

Having established the value of comparison over evaluation, and having outlined the ideas behind our comparison strategy, we now describe the actual comparisons to be used.

One of the most important but least tangible factors in the programming process is the "mental set" of the programmer: his view of the strengths, weaknesses, peculiarities, and proper use of the programming tools that he uses. Many factors are involved in the evolution of a particular programmer's mental image of the programming process; we shall try to isolate the ones that are affected by language.

When a programmer learns a language and continues to use it, his mind set is largely determined by that language. When a programmer learns to program in one language and then switches to another, his mind set is a juxtaposition of the two, and to a certain extent he will be programming mentally in one language and translating into the other as the code is written down. For our purposes, then, the mental language is much more important than the actual language.

We shall compare the effects that language-induced mind sets have on programming "in the small", that is, on the use of individual statements, statement-groups, loops, procedures, or even programs. For example, we compare how the language syntax and lexical structure shape the code that is written. We examine how the data abstractions that are built into the language affect the programmer's choice of representation and how the control structures provided by the language impact the grouping and local flow of the finished program.

We shall then compare the effects that languages have on programming "in the large", that is, the composition of individual programs and program segments into whole systems. Thus, we examine the ways a language can affect our ability to segment a task into independent subproblems, to assemble independent *components* into a system, and to maintain such a system once it exists.

Not all of these questions are meaningful to ask of all languages, and not all of the dimensions of comparison are relevant for all languages. But it is with these techniques and their implied goals that we examine the various language cores.

1.3. Scope of Project

We have undertaken only a comparison of languages. To use this comparison in a language selection process, one must consider many other factors of the programming environment. Although these aspects are often regarded as part of the language, they *in fact result from almost independent decisions* and consequently must be treated separately.

We have specifically ***excluded*** two major categories of potential comparison criteria from our study: compiler- or system-specific features and problem-domain issues. Thus, for example, we removed the Environment Division from our Cobol core because it is completely system-specific, and in some operating systems its functions have been subsumed by the job control and file system interface. Similarly, although we have discussed language features that affect modularity, we have not discussed non-linguistic problems of separate compilation and program linkage. Finally, such important, but problem-domain-specific facilities as Cobol's formatting facilities or Fortran's access to a scientific subroutine library are not considered.

The omission of these issues from this report is not to imply that we regard them as unimportant. On the contrary, in many contexts they may be more important than the issues considered here. They are omitted simply because they are outside the scope of the current study.

The original objectives of this study were:

to identify language features currently in use, and to evaluate them in terms of their specification and scope of effects, their use in practice, the provability of program correctness, and their suitability to programmers' needs;

to enumerate language features conducive and detrimental to good program structure, and to identify small variations that may have significant impact.

As the study proceeded, it became apparent that overall program organization and the way language features are used have a much greater impact on the quality of a program than does the selection or nonselection of individual features. We therefore adopted the "language-core" methodology described above. Evaluations of language features appear throughout the report, but we have made no attempt to construct exhaustive lists. Indeed, we believe that such lists would detract from the major results of the study.

Chapter 2
Definitions of Core Languages

In this chapter we define the "cores" of the four languages under investigation. A section is devoted to each; except for Ironman, the definitions of the cores are structured similarly. The common organization is:

- Philosophy
- Core Properties
    - Lexical and Syntactic Issues
    - Data and Data-Structure Issues
    - Control Structure Issues
    - Other Issues
- Definition of a Core
    - Syntactic Definition
    - Semantic Comments
    - Example Program

The section on "Philosophy" examines some of the contextual information surrounding the language definition. In many cases the essential flavor of a language derives from such information as the nature of the problem domain, the state of compiler technology at the time of the language definition, the typical mode of computer usage, and so on. This section attempts to capture those aspects of the context which strongly affected the language.

The section on "Core Properties" isolates those properties which *every* core of the language should contain. These are the features and policies that give the language its identity. The description of these properties is intentionally terse and hence depends to some extent upon the general knowledge of the reader.

The final section, "Definition of a Core", defines one possible core which reflects the properties outlined in the previous section. This core is defined syntactically, semantically, and by example. The syntactic definition uses an extended BNF notation (see below). The semantic definition is in prose; it is not intended to be complete, but rather to emphasize those aspects of the semantics which are most relevant to the subsequent comparison sections. The example program is an implementation of the same simple algorithm in each of the core languages. The primary purpose of the example is to illustrate that the cores do indeed capture the essence of the language. A specification of the example is given later in this section.

The syntactic definition of the cores is conventional BNF with the following additions and conventions:

1. Key-words (reserved words) are denoted by upper-case Roman letters.

2. Metasymbols are denoted by lower-case italic letters and may contain a hyphen, "-", to separate semantically significant words.

3. Where the intent of a meta-variable is obvious from its name, e.g., *identifier* or *identifier-list*, its definition is omitted.

4. The symbols { and } are meta-brackets and are simply used to group constructs in the meta-notation.

5. Three superscript characters, possibly in combination with a subscript character, are used to denote the repetition of a construct (or group of constructs enclosed in {}). In particular:
   "*" denotes "zero or more repetitions of"
   "+" denotes "one or more repetitions of"
   "#" denotes "precisely zero or one instances of".
   Since it is often convenient to denote lists of things separated by some punctuation mark, we denote this by placing the punctuation mark directly below the repetition character. Thus,

vvv ::= $a\{b \mid c\}$	defines a vvv to be an a followed by either a b or a c.
xxx ::= a^*	*defines an xxx to be a sequence of zero or more a's.*
yyy ::= $a\ b,^*$	defines a yyy to be an a followed by zero or more b's separated by commas.
zzz ::= $\{a \mid b\};^+$	defines a zzz to be a sequence of one or more things separated by semicolons -- where the "things" may be either a's or b's.

The problem chosen for the examples is based on [IBM 61a], pp. 79ff. The task is to take a number of input records which give an age and a salary, and for each 5-year age group to compute the average salary for the sample data. The algorithm is compactly represented in the following informal program:

```
people[all groups]←0;
earnings[all groups]←0;
for each input-record do
     begin
     add 1 to people[group];
     add salary to earnings[group];
     end;
for each group do
     begin
     if anyone in group
          then  average[group] ← earnings[group]/people[group]
          else  average[group] ← 0;
     end;
```

This program was chosen because it appeared to be suitable for all three of the existing languages undertaken in this comparison. Note, however, that the intent is that each of the programs be an exemplary use, *not* a typical use of each language; the languages were designed for different application domains and one cannot expect a single task to be typical for all of them. Also, since I/O is not a part of standard Jovial and had been excluded from the Fortran core, those two implementations were coded as subroutines which received their data from a main program. A complete Cobol program is presented, however. In Cobol the secondary-storage philosophy is very strong and independent subprograms are not possible; we felt that the example should reflect these properties.

2.1. Fortran Core

Fortran was one of the earliest "higher-level" languages for expressing numeric computations. It was developed in the mid 1950's, primarily for IBM equipment, and has been revised and extended numerous times. It is undoubtedly the most widely used programming language for scientific and numeric computing.

2.1.1 Fortran Core Philosophy

The fundamental style of Fortran was (and is) derived from the the requirements of numeric, scientific computing; thus it is dominated by those facilities needed for the manipulation of scalar values and homogeneous, rectangular, N-dimensional arrays of scalars. Thus, for example, the language has a natural notation for the evaluation of algebraic formulae, but not for manipulating symbolic data such as strings. Similarly, the iteration construct (DO) provides sequencing over the integers, reflecting the need to traverse arrays whose indices are integers.

The fact that Fortran was one of the earliest algebraic languages (early memos date from 1954) is apparent in several aspects of the language. These aspects still substantially impact the form of programs written in the language. For example,

1. The syntax of the language is "flat". Except in arithmetic expressions, nesting of syntactic constructs is generally not allowed. The most notable example of this is the ***statement***, which is conceptually (if not always physically) a single line of text. Statements may be independently grouped only by using the sub-program facility (SUBROUTINEs and FUNCTIONs).

2. The scope rules influence how programs, particularly large programs, are constructed. Variable names are generally local to a subroutine. The names of subroutines, on the other hand, are global to the set of modules which is linked together. Data from one subprogram is made available to another either by explicit parameter passing or by importing lists of global variables (COMMON).

3. Although Fortran is generally considered "machine independent", and it has certainly been implemented on a large number of dissimilar machines, there is a strong von-Neumann machine model evident in the language design. Some aspects of this model which impact program construction include:

   a. The mapping from an N-dimensional subscripted array to an address in a linear-contiguous address space is explicit in the language definition and is tailored for machines with common index-register properties.

   b. Allocation of storage for variables is static (i.e., determined at compile time).

   c. The primitive data types are precisely those supported by typical hardware. The selection of these types was aimed at granting the programmer convenient access to the hardware operations rather than at checking the consistency of a programmer's use of a particular variable.

Input/output has a distinctive philosophy, which is characterized by its record-orientation, by the existence of only a few primitive operations (READ, WRITE, REWIND, etc.), and by a fixed number of primitive mappings (as used in the FORMAT statements). However, the language retains its essential flavor if input/output is ignored, so we shall not include it in the core.

2.1.2 Fortran Core Language Criteria

2.1.2.1 Lexical and Syntactic issues

Spaces within a statement are not significant; they are neither necessary (they are not used as separators, so compressions such as "DO10I=1,N" are legal) nor prohibited (spaces may appear within an identifier, for example).

The initial orientation to punched cards has resulted in restrictions on which "columns" of the input line may contain the text of a statement.

If a variable has not been explicitly declared, it is implicitly declared when it is used in a statement. The type of an implicitly declared variable is determined by the initial letter of its name.

The statement organization is "flat". There is no syntactic nesting of statements and statements are identified solely by their statement label. By cultural convention, programmers do not indent Fortran programs. Although nothing prevents their doing so, Fortran programmers do not use indentation to provide visual cues. As a result, Fortran programs are vertically "flat".

2.1.2.2 Data Issues

The basic data types are scalar numeric values. The set of types is determined first by the Fortran standard [Fortran 66, Fortran 76] and then by a particular implementor (who may choose to implement only a subset of the types or to provide types which are nonstandard extensions). The set of types in any implementation is not extensible by the programmer.

Fortran allows the user to define aggregates of types. These aggregates are limited in form to rectangular arrays of scalar values, all elements of which must be of the same type (homogeneous).

All operations, except for parameter passing, are defined only for scalar operands, either simple variables or elements of arrays.

The enforcement of the concept of "type" is weak. There is no type-checking or conversion of values during subroutine calls, nor can there be. Within a module, facilities which are part of the language allow a user to specify several names (aliases) for a single storage location, and the type of the value in that location can be interpreted in several ways, depending upon the type of the alias used to access it.

The physical representation of data in an array (i.e., the mapping between subscript values and storage units) is explicitly visible to programmer and specified exactly by the standard ([Fortran 66] section 7.2.1.1.1, [Fortran 76] section 5.4.3).

The rules governing the set of variables which can be accessed at a given point in a program, i.e., the rules on scope and parameter-passing, are strongly characteristic of Fortran:

- Variable names are completely local to a program unit.
- Subprogram name and COMMON labels are global to all program units.
- Parameters provide for dynamic association of local names and external storage segments.
- COMMON provides for a static mapping between local names and external storage segments.
- The type interpretation of a mapped storage segment is determined by the type of the local name mapped to it.

Fortran was defined before any form of dynamic storage allocation was common, even the relatively inexpensive stack-oriented allocation found in almost all newer languages. This leads to some characteristic properties of Fortran:

- The physical representation of data is tied to its abstract representation.
- Storage requirements are fixed at coding time.

2.1.2.3 Control Issues

The basic control constructs are few in number: there is a simple IF, an integer-counting loop (the DO), and a GOTO. Although there are additional facilities in the form of assigned and computed GOTO's, these have much less impact on the form of Fortran programs than the more common constructs; they are therefore excluded from the core. Typically Fortran programmers synthesize their more sophisticated control from the three primitives.

Subprograms in Fortran may not be recursive. Parameters are either explicit or communicated through global (COMMON) storage. Separate compilation of Fortran subprograms has always been a standard feature and is heavily used; selective sharing of some global variables is supported via named COMMON.

2.1.3 A Fortran Core Language

2.1.3.1 Syntax

declaration ::=

SUBROUTINE *name* { (*variable* $,^{+}$) }*
| COMMON { /*name*/ }* *const-entity*$,^{+}$
| EQUIVALENCE { (*const-entity*, *const-entity*$,^{+}$) }$^{+}$
| *type* *const-entity*$,^{+}$

statement ::=

entity = *expression*
| GOTO *statement-label*
| IF (*boolean-expression*) *statement*
| DO *statement-label* *name* = *int-const-var*, *int-const-var* { , *int-const-var* }*
| CALL *name* { (*expression* $,^{+}$) }*

int-const-var ::= *integer-variable* | *integer-constant*
type ::= DIMENSION | REAL | INTEGER | LOGICAL
const-entity ::= *name* | *name* (*integer*$,^{+}$)
entity ::= *name* | *name* (*integer-expression*$,^{+}$)

Note that our syntactic definition ignores the statement/line orientation of Fortran as well as the placement of numeric statement-labels in colums 1-6, the continuation symbol convention, and the comment convention (a "C" in column one). All of these could have been expressed at considerable cost in the readability of the definition; we consider the price to be excessive and have therefore elided such details.

It should also be noted that there are two types, DOUBLE PRECISION and COMPLEX, in full Fortran which are not in the standard subset language. The new standard [Fortran 76] defines a new type, CHARACTER, as one of the basic types. The language is not altered significantly by the deletion of these types from its core.

2.1.3.2 Semantics Comments

The basic view of a Fortran program is as a collection of "program units", specifically a main program and a set of sub-programs. A program may be compiled as a single compilation unit or in a series of unrelated compilations; the language does not impose any constraints on the lexical or temporal sequence of the compilations, and only insists that the output of such compilations be available at the time a program is formed (e.g., linked) from its program units. Each program unit (module) is self-contained. Thus all variable names and statement labels are local to each module, and may not be referenced by any other module.

The assignment statement permits the result of an expression to be assigned to a variable or array element (generically referred to in the standard as an "entity"). All numeric results will be converted to the type of the entity to which the result is assigned. A logical result cannot be converted to a numeric result.

A GOTO statement may transfer control to almost any statement label in the module. Specific qualifications apply to inactive DO-loops and extended DO-ranges (where the text of the loop body is outside the range of the DO statement, e.g. [Fortran 66] section 7.1.2.8.2). A GOTO may not transfer control to any statement not in the module.[4]

The IF statement conditionally executes a single statement on the same line as the IF. If the boolean condition is false, control passes directly to the next executable statement in the program. Thus, the character of the IF is that either it executes one statement or it does nothing.

Fortran has a single iteration control construct, the DO-statement. The body of the DO-statement is delimited by a statement whose label is given in the DO-statement. Thus, the body of a DO-loop is (usually) contained entirely between the DO-statement and the statement whose label it mentions. (The notable exception, which has already been mentioned, is the "extended range" in which the statements that are logically part of the loop are located physically outside it.)

The SUBROUTINE declaration and CALL statements allow one program unit to invoke and to pass arguments to another. The CALL statement causes the arguments "to become associated" ([Fortran 76] section 15.6.2.2) with the formal (dummy) arguments of the SUBROUTINE declaration. This is usually accomplished by either call-by-reference or call-by-value-result.[5] The specific means by which this association is to be established and maintained are not explicitly stated, but a number of restrictions in the language are intended to permit various implementations of the parameter association to

[4] Note that it is possible to use certain facilities not in the core language, specifically the ASSIGN statement and the assigned GOTO to defeat this restriction. This technique is specifically prohibited by the proposed standard [Fortran 76] in section 15.9.4.

[5] Here and in the sequel we shall refer to four parameter-passing mechanisms: call-by-value, call-by-result, call-by-value-result,and call-by-reference. When a parameter is passed by value, the formal parameter is treated as data local to the called procedure (subroutine) which is initialized to the current value of the actual parameter. When a parameter is passed by result, the formal parameter is treated as data local to the called *procedure; upon completion of the procedure* call, the final value of this local variable is assigned to the actual parameter. When a parameter is passed by value-result, both this initialization and final assignment take place. When a parameter is passed by reference the address of the actual parameter is passed to the procedure, thus both accesses and assignments in the procedure have an immediate effect on the actual parameter.

behave in a consistent manner. Unfortunately, these cannot be enforced by the compiler (c.f., [Fortran 76] section 15.9.3.5; [Fortran 66] section 8.4.2).

A particular feature of the Fortran-parameter association mechanism is that there is no type-checking or conversion across sub-program calls. Thus, the representation passed by the CALL statement is determined by the type of the argument in the calling program unit, while the interpretation of the representation in the called program unit is determined by the type of the argument in the called unit.

The data type declarations allow the user to declare the type of a symbol explicitly; otherwise the type is determined by the initial letter of the symbol. Except in subprograms, where the bounds of an array may be specified when it is passed as a parameter, the bounds of an array are fixed at the time the statement is compiled.

The COMMON declaration allows users to specify explicitly the organization of storage relative to some base address. The actual address of the storage segment is determined by a facility such as a linkage editor or loader, but all COMMON segments with the same name will have the same base address. Note that modules are not required to specify the variable names in COMMON in the same order, with the same names, or with the same types as any other module. The interpretation of the COMMON segment by each module depends on order and type of the COMMON definitions local to that module.

The basic purpose of the EQUIVALENCE statement is to specify storage sharing, i.e., to allow logically disjoint sections of a program to share physically identical memory, possibly with different mappings of type and structure. It also permits a programmer to bypass the implicit type conversions and treat a storage location as if it were any of the valid data types.

2.1.3.3 An Example

```
      SUBROUTINE AVGSAL(AGE,SALARY,LENGTH,AVERAG)
      INTEGER AGE(LENGTH), NUM(20)
      REAL SALARY(LENGTH), AVERAG(20)
      DO 1 IX=1,20
         NUM(IX)=0
  1      AVERAG(IX)=0.0
      DO 2 I=1,LENGTH
         IX=AGE(I)/5+1
         NUM(IX)=NUM(IX)+1
  2      AVERAG(IX)=AVERAG(IX)+SALARY(I)
      DO 3 IX=1,20
  3      IF (NUM(IX) .NE. 0) AVERAG(IX)=AVERAG(IX)/NUM(IX)
      END
```

2.2. Jovial Core

During the past eighteen years Jovial has been implemented on a variety of systems and in several versions. Here we describe a core for the Level I subset of the J73 Jovial language [RADC 76]; this dialect was chosen because it is considerably cleaner than the J3 version [Air Force 76], yet does no violence to the essence of the older versions that have dominated the use of the language.

2.2.1 Jovial Core Philosophy

The nature of the Jovial language derives from two chief sources:

1. ***Roots in Algol-58:*** The basic structure of declarations, control structure, and program syntax stem from the influence of the early Algol efforts. As one result, Jovial exhibits a block structure similar to that of more modern languages.

2. ***Application to Command and Control:*** Jovial has been used to implement large modular command and control systems from its very beginning. In response to the needs of this application domain, the

language has acquired practical means of low-level control, low-level data representation, sharing of definitions among the various modules of large systems, and special tools for data declaration.

These two sources result in a tension between clarity and simplicity on the one hand and pragmatic richness on the other. This tension is not uncommon in programming language design and the language may respond to it in different ways. Generally the Jovial response has been to permit low-level facilities which syntactically resemble high-level ones; thus, for example, a mechanism is provided* which permits addressing an arbitrary memory word, but the mechanism appears similar to the high-level data-accessing mechanism. The net result is that, if the mechanism is properly used, clarity is preserved even though safety is not ensured.

2.2.2 Jovial Core Language Criteria

2.2.2.1 Lexical and Syntactic issues

Jovial source programs are free-form streams of characters.[6] Comments can appear almost everywhere and are bracketed by double quotes. The syntactic form is basically block structured, with statements and declarations either ending in semicolons or surrounded by BEGIN and END. Several keywords in data declarations are single letters. A standard macro facility, called DEFINE, provides an abbreviation facility which, like subprograms, can be used to improve the readability of programs.

2.2.2.2 Data Issues

Jovial data structures are rich and weakly typed.[7] The most basic data declaration is the ITEM declaration, which declares a variable of a specific

[6] Jovial identifiers may be from two to thirty-one characters long. Single character identifiers are reserved symbols in the language -- an unusual convention.

[7] By "weakly typed" we mean that typing is used to determine operations, e.g., fixed or floating addition, and is checked for subprogram parameters, but the type-checking is easily and explicitly circumvented by mechanisms in the language.

type and, optionally, of at least a specific size (in bits for numbers and bytes for character strings). The basic scalar types are signed integer, unsigned integer, bits, character string, and floating point. A facility for giving names to integers supports a weak form of enumerated type. Implicit type conversions take place with little restriction. The macro facility can be used to give symbolic names to constants of any scalar type.

Tables of one or more entries can be declared. An entry in these tables can be either a single item or a structure of several items. Thus arrays of heterogeneous groups of scalar items can be declared. In each table declaration the programmer can specify dense, medium, or loose packing of items, thus exercising some control over the space-time tradeoff.

The length of each table must be specified at compile time. Allocation of storage to data is of three forms: (1) RESERVE is permanent and static, (2) IN is local to a single invocation of the declaring routine, and (3) BASED allocates no storage, but uses an explicitly computed unsigned integer as the address for each reference. RESERVE allocation is the default and allows preloading of data. BASED allocation appears to be potentially error-prone, yet its skilled use in conjunction with TABLE and DEFINE declarations allows flexible control over storage structure.

2.2.2.3 Control Issues

Jovial control structures include the basic forms common in Algol-like languages, including the *if-then-else*, *for*, and *while* constructs, and a slightly restricted *goto.*

The procedure call mechanism allows ITEM parameters to be passed by value and/or result.[8] The syntax for procedure declaration and call both make it clear whether a given formal or actual parameter is passed by value, result, or value-result. Tables are always passed by reference. Type checking of actual parameters is done only for ITEMs.

[8] See the earlier footnote in the Fortran core definition for a description of these parameter mechanisms.

2.2.2.4 Other

Jovial implementations must support separate compilation of modules. The COMPOOL facility allows data, procedures, and definitions to be shared among these modules. A COMPOOL is itself a separately compiled module that declares the data, procedures, and definitions to be shared by a system of modules. Each using module invokes these declarations by means of a COMPOOL directive.

Jovial also provides low-level control over data representation and accessing. Representation of a table entry can be specified down to the level of the word and bit. Similarly, BASED allocation of data allows the programmer to map a template data description (for either an item or table entry) onto an arbitrary address. Although potentially error-prone, these features allow the Jovial programmer explicit access to physical locations that may be system- or hardware-sensitive.

2.2.3 A Jovial Core Language

2.2.3.1 Syntax

declaration ::=

> *item-declaration*
> | TABLE *table-name* [{{*number* :}$^{\#}$ *number*}$_{,}^{+}$] {N | M
> | D}$^{\#}$ *entry-specifier*;
> | STATUS *status-list-name* { V(*status-constant*) }$_{,}^{+}$;
> | PROC *proc-name* { (*input-parm*$_{,}^{*}$ { : *output-parm*$_{,}^{+}$ }$^{\#}$) }$^{\#}$;
>   *statement*;
> | COMPOOL *compool-name* ; {
>   *declaration* | BEGIN *declaration*$_{;}^{*}$; END };

statement ::=

  statement-label : *statement*
  | BEGIN *statement*$^{+}$ END
  | *variable*,$^{+}$ = *expression* ;
  | GOTO *statement-name* ;
  | IF *expression* ; *statement* { ELSE *statement*}$^{\#}$
  | WHILE *expression* ; *statement*
  | FOR *variable* : *control-clause* ; *statement*
  | *proc-name* *actual-parameter-list*$^{\#}$;

item-declaration ::= ITEM *item-name* *item-description* { = *constant*,* }$^{\#}$;

item-description ::=

  C *size-specifier*$^{\#}$
  | F *mantissa-specifier*$^{\#}$ *exponent-specifier*$^{\#}$
  | { S | U } *size-specifier*$^{\#}$ *status-list-name*$^{\#}$
  | B *size-specifier*$^{\#}$

entry-specifier ::=

  item-description { = *constant*,*}$^{\#}$;
  | BEGIN *item-declaration*$^{+}$ END

control-clause ::= *expression* { BY *expression* }$^{\#}$ { WHILE *expression*}$^{\#}$

2.2.3.2 Semantics Comments

A software system written in Jovial consists of a series of modules. The main program resides in a PROGRAM module. The other procedures are declared in PROC modules. Definitions, including procedure headings, macros, and shared variable names, are declared in COMPOOL modules. While each module may be separately compiled, the COMPOOL facility allows the sharing of procedure formal parameter specifications as well as procedure names and of shared variable names and types as well as the names of blocks of shared data.

The ITEM declaration specifies the name, type, size, and preloaded value of a scalar. The TABLE declaration makes use of the ITEM declaration syntax to specify the name, type, size, and preloaded data of each field in a table entry. In addition, it specifies a dimension list (bound at compile time) and the density of packing of the fields. The STATUS list declaration associates a list of symbolic status names with a range of integers. These may be used to treat an integer item as an instance of an enumerated type, and the language encourages the programmer to do so. The type of such an item remains S or U (i.e., integer) and integer notation and operations may be used at any time.

One unusual feature of the procedure declaration is the way it distinguishes between value, result, and value-result for scalar formal parameters. Any scalar formal may appear once in the input parameter list, in which case it will be passed by value. A scalar formal may also appear once in the output parameter list, in which case it will be passed by result. Should a scalar formal appear in both lists, it is called by value-result. Thus in the procedure heading:

```
PROC MUNGE ( AA, BB : AA, CC )
```

BB is passed by value, CC is passed by result, and AA is passed by value-result. The syntax of the procedure call is similar and demands proper specification of the actual parameters. Thus in the procedure call:

```
MUNGE ( XX, YY : ZZ, XX );
```

YY is passed by value, ZZ is passed by result, and XX is passed by value-result. This control over parameter passing at both procedure declaration and call is remarkably simple and flexible, yet almost unique. Surprisingly, it is a product of Algol-58 from which Jovial was derived but has not been kept in other languages.

The assignment statement is quite ordinary; it should be observed, however, that implicit type conversions occur freely.

2.2.3.3 An Example

```
PROC AVGSALARY( EACH'DATA, AVERAGE, LENGTH );
  BEGIN
  TABLE EACH'DATA [1];
    BEGIN
    ITEM AGE S;               "THE AGE OF ONE PERSON   "
    ITEM SALARY F;            "AND HIS ANNUAL INCOME   "
    END "EACH'DATA"
```

```
TABLE AVERAGE [20]     F;      "AVERAGE ANNUAL INCOME OF AGE GROUP"
ITEM  LENGTH           S;      "NUMBER OF ENTRIES IN EACH'DATA"
TABLE NUM     [20]     F;      "NUMBER OF ENTRIES IN AN AGE GROUP"
ITEM  ENTRY            S;      "INDEXES INTO EACH'DATA "
ITEM  SLOT             S;      "INDEXES INTO AVERAGE AND NUM"

FOR SLOT:0 BY 1 WHILE SLOT<20;           "CLEAR NUM AND AVERAGE"
  NUM[SLOT], AVERAGE[SLOT] = 0.0;

FOR ENTRY:0 BY 1 WHILE ENTRY<LENGTH;  "DEVELOP SUMS OF INCOME"
  BEGIN
  SLOT=AGE[ENTRY]/5;
  AVERAGE[SLOT]=AVERAGE[SLOT]+SALARY[ENTRY];
  NUM[SLOT]=NUM[SLOT]+1.0;
  END

FOR SLOT:0 BY 1 WHILE SLOT<20;           "USE SUMS TO BUILD AVERAGES"
  IF NUM[SLOT]>0.0;
    AVERAGE[SLOT]=AVERAGE[SLOT]/NUM[SLOT];
END
```

2.3. Cobol Core

The design of Cobol was motivated by the need to write non-numeric file-oriented programs. Such programs involve substantial amounts of externally stored data; the programming problems are frequently dominated by the format-sensitivity of this data. The original design philosophy explicitly considered the issue of program compatibility and portability of data among machines. This led directly to the explicit separation of the algorithmic or procedural parts of a program from the description of the data involved. Further, information about the object machine was to be separated from both algorithm and data definitions [Cobol 60, Cobol 74].

2.3.1 Cobol Core Philosophy

Most programs in use today are written in Cobol; this is a tribute to the language design, the importance of the problem domain, and the standardization and promotional efforts of the U.S. government.

The language which has evolved from the original goals is dominated by the following characteristics.

1. ***Closeness to secondary storage:*** The Cobol programmer is in close contact with secondary storage. Windows into externally stored files are an integral part of his program's name space, and a substantial part of the programming task is concerned with defining the formats to be used for interpreting this data. The external files are viewed as sequences of unit records of characters, and interpretations are imposed by defining complex field, array, and record structures on the underlying character stream.

2. ***Similarity to English:*** The designers intended programs written in Cobol to be readable by people without knowledge of the object computer. They approached this goal by making the language resemble English as much as possible.

3. ***Multiplicity of Specific Features:*** The language comprises a large collection of individual features. Each of these either specifies some property of a data organization or operates on the data so defined. The feature collection is large enough (36 verbs and 262 keywords) to make the act of programming resemble selection from a menu of features more than synthesis from a set of abstract primitive operations.

4. ***Separation of concerns:*** Information related to algorithms, data definitions, and machine characteristics is viewed as separate and independent. The distinction is so strong that a top-level separation is made in every program.

2.3.2 Cobol Core Language Criteria

2.3.2.1 Lexical and Syntactic Issues

A Cobol program is a sequence of ***statements***, each beginning with a keyword. *Some statements are procedural, others are declarative, and some* declaration keywords are numeric, but the essential keyword-driven syntax *remains the same.* The rules for assembly of Cobol statements into a program are so complex that they could well be called a "vertical syntax". Cobol reference manuals are almost universally divided into two separate but interwoven pieces: one describes the syntax of the various statements, and the other describes the syntactic rules for combining them.

A Cobol *statement* is a verb followed by its operands or modifiers. A *sentence* is one or more *statements* terminated by a period. *Sentences* can be clustered into *paragraphs*, and *paragraphs* into *sections*. In divisions other than the PROCEDURE DIVISION, the notions of *sentence* and *paragraph* are vague. Within a *statement*, operands are separated by sequences of blanks and keywords; the programmer may use commas and semicolons as visual cues for the reader, but they will be ignored by the compiler. The syntax for some particular statement frequently requires noise words, whose presence is not needed to parse the statement unambiguously but which make the statement more readable to a layman. Thus, the Cobol statement

```
SORT X-FILE ON ASCENDING KEY PART-NO USING A-FILE GIVING
B-FILE.
```

could be reduced by removing the noisewords to

```
SORT X-FILE ASCENDING PART-NO A-FILE B-FILE.
```

without becoming uncompilable.

2.3.2.2 Data Issues

Cobol requires the programmer to specify both internal program data organization and file data, using almost, but not quite the same mechanism for both. The essence of Cobol data declarations is the superposition of structure on the underlying memory. File data declarations specify how the characters in the file are to be grouped together into data items, while data declarations for primary storage specify how the memory address space is to be organized into variables, arrays, and such.

A Cobol elementary datum, called an *item*, is the smallest unit of declaration. Cobol provides a set of item types, with variations, and the programmer may combine elementary item types into bigger structures in any combination.

However, the programmer cannot build a new item type. The set of item types provided by the language (DISPLAY, COMPUTATIONAL, etc.) is designed to be implementable in terms of data types supported by the underlying hardware, and there is a sufficiently rich and redundant collection of item types that widely varying hardware features can be used efficiently.

Cobol names are global to the entire program, but need not be globally unique: procedure and data names are qualified, where necessary, by the name of the section or data structure that contains them.

2.3.2.3 Control Issues

Cobol has two control constructs, GOTO and PERFORM. A GOTO is not in any way restricted as to destination. A PERFORM combines the functions of iteration and routine-call, with the property that the scope of an iteration is delimited by the PERFORM and not by any marks in the code being iterated. (This is in fact why the language can afford to allow arbitrary GOTO destinations, as there can be no interference with loop code). PERFORM implements for-loops, while-loops, and combinations thereof.

2.3.3 A Cobol Core Language

2.3.3.1 Syntax

Cobol program ::=
    ENVIRONMENT DIVISION. *environment-division-body*
    DATA DIVISION. *data-division-body*
    PROCEDURE DIVISION. *procedure-division-body*

data-division-body ::=
    FILE SECTION. *FD-paragraph*$^+$
    WORKING-STORAGE SECTION. *data-declaration-statement* .$^+$

FD-paragraph ::=
    FD *file-name* DATA RECORD IS *record-name.*
    01 *record-name* *item-description*

data-declaration-statement ::=
    01 *data-name* *item-description*
    | 77 *data-name* *item-description*
    | 88 *data-name* *88-description*

```
procedure-division-body ::= { procedure-label SECTION. procedure-paragraph+ }+

procedure paragraph ::= procedure-label. paragraph-body


paragraph-body ::=
        {       COMPUTE data-name = arithmetic-expression
                | ADD datum TO data-name
                | SUBTRACT datum FROM data-name
                | MULTIPLY datum BY data-name
                | DIVIDE datum INTO data-name
                | PERFORM procedure-label {iteration-clause}#
                | MOVE data-name TO data-name
                | IF condition THEN paragraph-body ELSE paragraph-body
                | input-output statement
        }.* { GOTO procedure-label  |  STOP RUN}#.
    | EXIT.
    | COPY library-entry-name.
```

```
item-description ::=
        attribute-description ,*.
          {larger-level-number  data-name  item-description}*.
      | COPY library-entry-name.

attribute-description ::=
        PICTURE picture-string
      | USAGE {COMPUTATIONAL | DISPLAY }
      | VALUE IS literal-constant
      | REDEFINES data-name
      | OCCURS integer-constant TIMES

input-output statement ::=
        OPEN { INPUT | OUTPUT } file-name ,+
      | CLOSE file-name ,+
      | READ file-name { AT END paragraph-body}#
      | WRITE record-name

data-name ::= declared-identifier

datum ::= data-name | literal-constant.

condition ::= Boolean-expression | 88-level-data-name

88-description ::= VALUE IS literal-constant
```

Note that the syntax of Cobol, like that of Fortran, is format-sensitive; i.e., it has specific rules for the placement of a statement on a card image. It does not seem productive to exhibit these requirements in the core; thus, the definitions ignore them.

Also note that the *environment-division-body* is not defined because it specifies highly machine-specific information; we feel that such details are not relevant to the core.

2.3.3.2 Semantics Comments

A Cobol program is divided into a data-description part and a algorithm-description part. It is largely from this separation that Cobol derives its noteworthy portability. The full language specification also contains an identification division and an environment division; these were excluded from the core as not being essential to the central language characteristics. In particular, most of the functions of the environment division have been subsumed by the operating system in modern implementations of Cobol.

Cobol contains, as a part of the data-declaration mechanism, the ability to specify implicit computations, i.e. operations performed indirectly as data is moved into a record. This unique feature helps to simplify the procedure division by removing from it the complex, detailed coding that would otherwise be required.

The character of Cobol hierarchical data description is difficult to capture in the syntax. A good mental picture is to see it as a stepwise decomposition of *a data record*, with each level specifying a more-detailed breakdown of the data in the previous level.

Note that the ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs have the same syntax, and store their result into the second argument; this construction leads to the clumsy DIVIDE...INTO sequence. We have excluded from our core Cobol the DIVIDE...BY sequence, which has the unusual property of storing into its divisor: DIVIDE X BY Y will set Y to the quotient X/Y.

The COPY construct allows a programmer to include library data-definitions or code paragraphs in his program. This simplifies the sharing of tapes among programs and supports common file-formats and algorithms.

2.3.3.3 An Example

In the following version of the example program we have excised certain machine-specific information. The program will not run without this information, but it is no more pertinent to the language comparison than, for example, the control commands used to run the program. To show where this information was omitted we have inserted lines containing the string "<machine specific information omitted>".

```
ENVIRONMENT DIVISION.

        <machine specific information omitted>

DATA DIVISION.
FILE SECTION.
FD      INPUT-DATA
                DATA RECORD IS INPUT-RECORD,
                <machine specific information omitted>
01      INPUT-RECORD USAGE IS DISPLAY-7.
        02      AGE     PICTURE 99.
        02      FILLER  PICTURE X.
        02      SALARY  PICTURE 99999.

FD      OUTPUT-DATA
                DATA RECORD IS OUTPUT-RECORD,
                <machine specific information omitted>
01      OUTPUT-RECORD USAGE IS DISPLAY-7.
        02      FILLER PICTURE X(132).

WORKING-STORAGE SECTION.
01      OUTPUT-LINE.
        02      AGE     PICTURE Z9.
        02      FILLER  PICTURE X VALUE IS "-".
        02      AGE-UPPER-LIMIT PICTURE Z9.
        02      FILLER PICTURE XXX VALUE IS "   ".
        02      SALARY  PICTURE ZZ,ZZ9.

01      DATA-BUCKETS.
        02      EACH-BUCKET OCCURS 20 TIMES.
                03      NUMBER-OF-PEOPLE PICTURE 9999.
                03      TOTAL-INCOME PICTURE 9999999.
                03      AVERAGE-INCOME PICTURE 9999999.
```

```
01      SCRATCH-VARIABLES.
        02      I                   PICTURE 99.
        02      AGE-VALUE           PICTURE 99.

77      BUCKET-LIMIT PICTURE 99 VALUE IS 20.
77      INPUT-OK PICTURE 9 VALUE IS 1.
77      INPUT-EOF PICTURE 9 VALUE IS 0.

01      GLOBAL-STATUS.
        02      INPUT-FILE-FLAG PICTURE 9.
        88      DONE-WITH-INPUT VALUE IS 0.

77      YEAR-SPAN PICTURE 99 VALUE IS 5.

PROCEDURE DIVISION.
MAIN SECTION.
START.
        PERFORM INITIALIZATION.
        PERFORM DATA-INPUT UNTIL DONE-WITH-INPUT.
        PERFORM AVERAGING.
        PERFORM DATA-OUTPUT.
        PERFORM TERMINATION.
        STOP RUN.

HOUSEKEEPING SECTION.
INITIALIZATION.
        OPEN INPUT INPUT-DATA.
        MOVE INPUT-OK TO INPUT-FILE-FLAG.
        OPEN OUTPUT OUTPUT-DATA.
        PERFORM ZERO-BUCKETS VARYING I FROM 1 BY 1 UNTIL I
                        GREATER THAN BUCKET-LIMIT.

TERMINATION.
        CLOSE INPUT-DATA, OUTPUT-DATA.

ZERO-BUCKETS.
        MOVE 0 TO NUMBER-OF-PEOPLE(I), TOTAL-INCOME(I).
```

```
DATA-INPUT SECTION.
READ-INPUT-DATA.
          READ INPUT-DATA, AT END
                    MOVE INPUT-EOF TO INPUT-FILE-FLAG.
                    GO TO END-OF-INPUT-DATA.
          COMPUTE I=AGE OF INPUT-RECORD / YEAR-SPAN + 1.
          ADD 1 to NUMBER-OF-PEOPLE(I).
          ADD SALARY OF INPUT-RECORD TO TOTAL-INCOME(I).

END-OF-INPUT-DATA.
          EXIT.

COMPUTATIONS SECTION.
AVERAGING.
          PERFORM AVERAGE-BUCKET VARYING I FROM 1 BY 1 UNTIL I
                              GREATER THAN BUCKET-LIMIT.
AVERAGE-BUCKET.
          IF NUMBER-OF-PEOPLE(I) NOT EQUAL 0
                    COMPUTE AVERAGE-INCOME(I) = TOTAL-INCOME(I) /
                              NUMBER-OF-PEOPLE(I).

RESULT-DISPLAY SECTION.
DATA-OUTPUT.
          PERFORM DO-OUTPUT VARYING I FROM 1 BY 1 UNTIL I GREATER
                              THAN BUCKET-LIMIT.

DO-OUTPUT.
          COMPUTE AGE-VALUE  = YEAR-SPAN * ( I - 1 ).
          MOVE AGE-VALUE TO AGE OF OUTPUT-LINE.
          COMPUTE AGE-VALUE = AGE-VALUE + YEAR-SPAN - 1.
          MOVE AGE-VALUE TO AGE-UPPER-LIMIT.
          MOVE AVERAGE-INCOME(I) TO SALARY OF OUTPUT-LINE.
          MOVE OUTPUT-LINE TO OUTPUT-RECORD.
          WRITE OUTPUT-RECORD.
```

2.4. Ironman Core

Identifying an Ironman core presents a somewhat different task from the previous definitions. We have complete language specifications and a body of user experience for Fortran, Cobol, and Jovial, but for these languages we must deduce the intentions of the designers from the legacy of features in the languages. In contrast, for Ironman we have an explicit statement of goals and philosophy [Ironman 77], but no language designed to meet those goals (nor, for that matter, do we have a guarantee that it is possible to design a language to satisfy the goals). However, the general design criteria, together with requirements for specific features, do yield an image which will probably be preserved by any satisfactory candidate for the language.

2.4.1 Ironman Core Philosophy

The attitudes which emerge are dominated by two practical considerations:

1. The intended ***application to embedded systems*** leads to (restricted) generality and specific machine-relevant considerations.

2. The importance of ***maintainability*** leads to a spirit of clarity, simplicity, and understandability.

The dominant characteristics of the Ironman language seem to be:

1. ***Simplicity of language:*** This is explicit in the goals, and it is supported by lexical, syntactic, and semantic considerations in the specific requirements. The simplicity of the language is intended to arise from the similarity of related constructs, and from uniform assumptions about the interpretation of program statements. The simplicity is further supported by a number of requirements for compile-time checking.

2. ***Rich data structuring:*** Data structuring facilities include not only scalar values and structures which are aggregates of scalars, but also complex record structures and mechanisms for defining application-specific data organizations.

3. *Programmer-defined types:* A powerful abstraction facility is intended to allow definitions of data structures and related operations to be encapsulated and protected. Abstractions defined by this mechanism are to have same status as the primitive data abstractions (types) of the languages.

4. *Access to underlying hardware:* Explicit provisions are made for providing access to all the features of the underlying hardware and for interposing as little language overhead as possible. In this way the programmer can obtain very detailed control over efficiency of both code and data representations.

5. *Conscious restraint:* A number of facilities have been omitted and the power of others has been restricted in order to retain leanness and safety in the language. In addition, the requirements show considerable sensitivity to the usability of the language.

We note that the view projected by the Ironman requirements is not totally self-consistent, and it may not be possible for any language to simultaneously satisfy all the goals. In particular, there seems to be a conflict between a desire for readability and safety on the one hand and detailed control over access to the underlying hardware on the other.

2.4.2 Ironman Core Language Criteria

We shall now address the Ironman requirements along traditional lines and show how they can be expected to lead to a language with the philosophy described above. Each of the entries in this sketch is supported by references to specific requirement numbers in the Ironman report [Ironman 77].

2.4.2.1 Lexical and Syntactic Issues

The general design criteria call for simple syntax both explicitly (1E) and implicitly, in support of maintainability (1C) and formal definability (1H). The restriction to only the necessary generality (1A) also supports simplicity.

Specific lexical requirements include the ability to use a 64-character alphabet (2A), no special interpretation of line boundaries (2D), the ability to use mnemonic identifiers (2E), restriction of reserved words to be delimiters (2F), and built-in numeric and string literals (2G,2H).

The syntax is required to be regular and free-form (2B), and similar notations are required to denote similar constructs (2D,4A). The regularity extends to the use of expressions, which can appear wherever constants and variables of similar type are allowed (4D). Regularity is extended to programmer-defined types (3C,4A); they must be treated on a par with the basic types. The complexity of the language is held down by the requirement that the built-in control statements be of minimal number and complexity (6A). Defaults are forbidden (1C,5B,5E), as is extension of the lexical and syntactic structure of the language (2C,3-2A,6A).

2.4.2.2 Data Issues

The data definition facilities of Ironman are dominated by strong typing, rich heterogeneous nonscalar structure, and provisions for programmer extension of the type system. Access to the object machine is provided by tailoring the primitive arithmetic types to the available precision, by allowing programmers to specify object representations, and by providing a type extension mechanism. These facilities are motivated by concerns for maintainability (1C), efficiency (1D), and implementability (1F).

Strong typing is supported by requiring compile-time type checking (3A,13D), requiring safe treatment of variant records (3-3H), and prohibiting implicit conversions between types (3B,3-1E,3-1H).[9] The type philosophy is carried into the rules for interpreting expressions; the syntax of expressions is independent of operand type (4A), and their types are determinable at compile time (4B).

The requirements for rich data structures are most apparent in section 3, which requires the language to support heterogeneous composite types (3.3),

[9] Note that the rule on implicit conversions may be in conflict with a requirement for operations between types (3-5D); also, floating point precisions are not treated as distinct types (3-1C). However, mixed-mode arithmetic appears to be allowable under criterion 1C.

enumeration types including Booleans (3.2.1), characters (3.2.2), and programmer-defined enumerations (3-2A,3-2B), and sets of enumerations (3.4). The composite types may be recursively defined (3-3A); new types shall automatically have field accessing operations (3-3C,5F), value constructors (3-3C), assignment (3-3C, 5F), and (for scalars) equality test (3-2A). Further, arrays shall have built-in operations on contiguous subarrays (3-3E), the subscripts for arrays may be any enumerated type (3-3D), variants shall be supported (3-3H), and assignment shall be permitted between records with the same structure (3-3F). Types are not to be fully general; procedures, functions, types, labels, statements, and exceptions shall not be treated as types (5D).

Ironman will support programmer definition of data abstractions in a number of ways. In addition to composing data definitions, it shall be possible to define new types (3C) and to abbreviate specification phrases (5G). Programmer-defined types will, as far as possible, have the same status as built-in types (3C,7A). This will not be entirely possible; for example, new literal values may not be added (2B,2.2,3-2A), constants are not permitted for dynamically allocated types (5A), and it is unlikely that the requirement for evaluating constant expressions before execution time will be implementable (4E). The new types may be defined by enumeration (3.2), or they may be encapsulated with private data (3.5). In addition, types (as well as functions and procedures) may be defined generically (12D).

The requirements for numeric types reflect a concern for allowing good use of the underlying hardware. Fixed-point arithmetic (3.1.2) is important in many applications, and the implementation of floating-point arithmetic is encouraged to match the available machine precision (3-1C,3-1D). The programmer may control the physical representation of his data (11A), declare properties of the object machine (11C), insert encapsulated machine code (11E), and write programs whose behavior depends on the object hardware configuration.

2.4.2.3 Control Issues

Ironman control is dominated by the philosophy on simplicity and restraint. The design goals call for readability (1C), efficiency (1D), avoiding of error-prone features (1B), and formal definability (1H) lead to the choices of both local and nonlocal control constructs and to the restrictions placed on the features selected.

The local control mechanisms comprise the usual nestable, structured statements plus the goto (6A,6B,6C,6E,6G). Attention to safety is evidenced in the restrictions on destinations of gotos (6G) and the protection of loop control variables (6F). This concern fails, however, in the interaction of short circuit evaluation for *certain* control structures (6D) with the possibility of side effects (4C).

The nonlocal control mechanisms address functions and procedures (7), parallel processing (9), and exception handling (10). Procedures and functions may be invoked and defined recursively (7B). They are constrained by requirements on strong typing (7D,7G,7H), parameter binding rules (7E,7F), and aliasing (7I). In addition, functions may not use variables from the caller's scope (7E); note that this does not completely preclude side effects (4C). A number of facilities for parallel processing (9) and exception handling (10) are required by the report, but few details are given. Since current software engineering practice has not achieved consensus on these topics, we cannot predict the form the features will take.

2.4.2.4 Other

The philosophy and design goals demand that the object machine be made accessible to the programmer (1D,1F). The primary mechanisms for providing this access are input-output (8A,8B), a way to interrogate and interact with the machine configuration (8E,11C,11D), provisions for specifying timing properties (9D,9E,9F), control over the physical representation of data (11A), and machine language insertions (11E). The power of these mechanisms shall be circumscribed in order to preserve machine independence (1G). The primary techniques for enforcing these restrictions are various kinds of encapsulation (11D,11E).

2.4.3 Ambiguities and Conflicts

Some of the requirements appear to be either mutually contradictory or substantially at variance with the language philosophy and general design criteria. The suspect requirements include those dealing with scope rules, the notion of type, dynamically allocated variables, and the simultaneous requirements for machine independence and detailed access to the object machine.

Other requirements, such as those for parallel processing, exception handling, time-space tradeoffs, and the complete integration of programmer-defined types in the type structure, may or may not be possible to satisfy.

In other languages, extensions and modifications may compensate for unevenness in the original design. However, this escape mechanism is not available to Ironman, which shall have no subsets or supersets (13C).

These issues are discussed in the sections below. In many cases, the inconsistencies involve many sections of the report. Only the most significant are mentioned here; the comprehensive web of supporting points is listed in appendix A.

2.4.3.1 Scope Rules

Although it is clear that the designers intend to have a simple, safe scope rule, the specific requirements are at least unclear and possibly incompatible. Most of the statements in the report support the view of a nested, block-structured language with lexically static inheritance of names and other restrictions intended to preclude side effects. However, this basic model may not be compatible with the encapsulation mechanism (3-5B,3-5C,3-5D,6G), the availability of own variables (3-5C,4C,7E,11E), dynamically allocatable variables (3-3J), the exception mechanism (10C), explicit importation and exportation of definitions (3-5C,12B), and embedded machine code (11E).

The most significant potential difficulty is that it may be possible to defeat the scope rules by passing information through hidden paths (3-5C,10C,11E,12B), thereby circumventing the design criteria of readability, maintainability, simplicity and reliability. Such hidden paths can reasonably arise through private data of encapsulations (3-5C) and through explicit importation and exportation of names between modules (12B). If dynamically allocated variables can be used to construct objects with shared substructure (3.3.3) the resulting aliasing produces hidden paths. Both exceptions (10C) and machine code (11E) may also permit communicating in non-nested ways. Certainly these mechanisms may be used safely, and the name or data sharing may be explicit and well-defined, but the policies which are being used in any particular program can be much more complex than seems intended by the philosophy.

2.4.3.2 Types

Further difficulties may arise in determining when two variables or expressions are of the same type and in treating programmer-defined types as if they were primitive.

Strong type checking requires precise determination of the type of each variable or expression. This is clearly intended (1F,3A,4B,7G,8C,12B), but it is not clear exactly when two variables are of the same type. For example, the descriptions of range, precision, and scale specifications suggest that these specifications do not affect the types of variables (3-1C,3-1F). However, other requirements appear to discriminate on the basis of these specifications (3-1H,3-3B,7G).

More generally, it is not clear whether two variables are of the same type when the types used in their declaration are identical in name or only when those type definitions have identical (or even similar) structure. The first position is consistent with the philosophy (1B,1F) and certain requirements (3B,3-3D,7H). The second is suggested by other requirements (3.2,3-2B,3.3,3-3F,3-3H,3-5A,11B). In particular, it appears that partial record copies may be required (3-3F). Dynamic substructure of dynamically allocated types is simply inconsistent with other requirements (3-3I). In addition, the type structure can be violated under certain circumstances (3-5D,11E).

A second set of unclear requirements concern the use of programmer-defined types. The interaction of such definitions with scope requirements was discussed above, but problems may also arise in the interaction with the type structure. In particular, although any restrictions on defined types must apply to all types (3C) and all restrictions must be enforceable by the compiler (1F), there may be problems with enforcing these standards for literals (2.2,3-2A), compile-time evaluation of constant expressions (4E), and certain "automatically-defined" operators (3-2A,5F,7A).

2.4.3.3 Dynamic Allocation

Ironman is required to support types whose elements are dynamically allocated. This will require a feature-specific mechanism that is unlike anything else in the language (3-3I). If, as suggested, these types are required by the applications and if they can be incorporated in the language tastefully, the result may be consistent (1A,1D). However, the distinction between these types and composite types (3-3I,5A) is at variance with the philosophy of consistency (1E, etc.); it produces a new kind of scope rule very different from the dominant rule (3-3J), and the requirement implies a garbage-collected allocation strategy which may be more general (less efficient) than necessary for particular applications (1A).

2.4.3.4 Parallel Processing and Exception Handling

As discussed in section 2.4.2.3, the requirements for parallel processing (9) and exception handling (10) are qualitatively different from the remainder of the report. There is more than one current proposal on how best to handle each of the issues, and none has yet been recognized as dominant. In addition, it is curious that "real time" processing appears under the heading "parallel processing" (9D,9E).

Even the statements of these requirements have a different flavor from the rest of the report. The interactions between the specific requirements (9,10) and the remainder of the report are only

1A: Embedded applications require real time control, self diagnostics, and parallel processing.

1G: The requirement (9E) for a real time clock is at variance with 1G, which demands that the language not dictate the character of the object machine.

3-1A: Numeric overflows create exceptional conditions.

6G: The goto statement shall not transfer control out of parallel control structures.

The requirements do, however, ask only for a safe, relatively well-understood subset of the current proposals. hopefully this can be done in a manner which will not interfere with future developments. It is not clear, however, whether such a "safe" subset exists -- or, if it does, whether it is adequate to cover the envisioned need. Features such as exception handling may be integral to the language design philospy and may not be easy to "patch-on" later.

2.4.3.5 Time-Space Tradeoffs

Ironman is required to permit the programmer to exercise control over time-space tradeoffs. This is evidently to be accomplished in two ways: automatically, by stating criteria to control which resources are to be optimized (1D, 11F, 13A) and manually, by explicitly defining representations and policies (8E, 11A, 11B, 11E).

"Automatic" control may be achievable to the extent of allowing the programmer to decide whether packed or unpacked representations are to be used and whether the compiler should optimize the object code. However, we see no way to provide a continuum of choices or arbitrary objective functions.

It is clearly possible to permit "manual" control of the tradeoff. The facilities provided for this purpose will probably be powerful enough to interfere with other language goals. We mention some of the potential problem areas elsewhere.

2.4.3.6 *Hardware Access vs. Machine Independence*

The fundamental tension between machine access and independence is captured in criterion 1G: "The language shall strive for machine independence. It shall not dictate the characteristics of object machines or operating systems... There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration..."

The relevant requirements address issues ranging from the availability of special features (1A, 8A, 10A) to facilities for using the hardware effectively without special effort (1D, 3-1C, 3-1D, 11A) to safety provisions (1C, 8E, 11C, 11D, 11E).

It seems reasonable to expect that the needs of embedded computer applications can be satisfied only by providing an escape mechanism that allows the programmer to interact with the individual properties of each computer. Indeed, it would be unwise to attempt to anticipate such individual characteristics in a high-order-language design. The escape mechanism does, however, require special attention to ensure its consistency with the remainder of the language.

Chapter 3
Comparisons of Core Languages

Section 1.2 outlined a number of dimensions along which languages can be compared. In this chapter we perform those comparisons for the core languages defined in Chapter 2. Note that the comparisons are directed at the cores, not at the full languages.

3.1. Comparison of Philosophies

Language designers, we believe, make a number of fundamental philosophical decisions which significantly affect the nature of the language they produce. While these decisions are not always explicit (they may be the result of historical or other contextual factors), they tend to have an enormous impact. Thus, we shall begin our comparison by examining some of these issues for the four languages in question.

The issues we shall consider are:

1. *The Designer's View of the Problem:* The designer's image of the nature of the problem domain is undoubtedly the most important of all influences on the language. The distinction between numeric computations, data processing, and command-and-control is the most obvious manifestation of this view. In addition, however, whether the designers perceived solutions to problems as consisting of single programs, sets of single programs, or large systems is a major factor.

2. *Selection vs. Synthesis:* There is a spectrum of approaches to providing "power" in a programming language. At one end of this spectrum is the selection, or menu, approach: to anticipate every possible need in the form of specific features, thus making the programming task one of simply selecting the right option for a given problem. The other extreme of this spectrum is the synthetic approach: to supply a small set of basic mechanisms together with a

flexible means of composing them into larger units, thus making the programming task one of constructing the proper substructure in which ultimately to state the solution to a problem.

3. ***The Degree of Permissiveness:*** Another important spectrum in language design is the degree to which it ***enforces, encourages, permits***, or ***prevents*** various programming practices. In software, just as in other technologies, powerful tools are often dangerous. Precisely the same language features which are touted as powerful are often subject to misuse. The enforcement spectrum, then, expresses the degree of permissiveness in the language relative to the use of these powerful, but dangerous mechanisms. A given language generally takes a stand at some point on this spectrum and then must respond by either (1) imposing restrictions on the power of the language and then permitting controlled circumvention of its own restrictions, or (2) providing powerful but dangerous features and allowing conventions to be imposed on their use.

Not all combinations of these philosophical positions make sense, nor are any of the languages in question completely consistent in their positions on these issues. Nevertheless, observing the way the languages respond to these questions provides insight into their actual use.

3.1.1 The Designer's View of the Problem Domain

The major questions here are the designer's image of the problem domain, and the kinds of programs which are likely to provide solutions to problems in that domain. As noted above the most obvious distinctions are the traditional "scientific" vs. "business", etc. These simple distinctions, however, belie many of the really important issues. Cobol, for example, is not influenced merely by the fact that it is aimed at business data processing, but also by a 1950-style view of the nature of the solution to business data processing problems -- a view that was dominated by manual and/or unit-record techniques.

Fortran: Fortran is, of course, dominated by its orientation to scientific, numeric computing. Even though it was one of the earliest high-level programming languages, much of the early history of computers revolved around just the kind of numeric computing that Fortran is designed to do. Thus the needs of this class of users was well understood. The early definition of the language is manifest in other, sometimes subtle ways, however. The most obvious is its quaint syntax. A more subtle one is its extreme concern with efficiency; in those early days, it must be remembered, most "large" users did not believe that *any* higher-level language could replace assembly language because the former were inherently too inefficient.

Cobol: The experience with manual and unit-record approaches to business data processing applicationsappears to pervade Cobol. The most natural programs in Cobol are those which serially process a sequence of identical records, performing the same computation on each record, and writing the results of that computation into another series of records. Even the laudable goal of readability was, we believe, influenced by this view of the nature of business data processing. In the manual and unit-record world a complex operation is divided into a series of highly standardized steps of the type described above; no individual step is very large. Within this context the notion of readability only applies to relatively small contexts -- and hence Cobol's preoccupation with English-like style for the individual "sentence".

Jovial: Command and control problems of the 1950's, exemplified by the SAGE effort, forced the development of large complex software systems. These systems needed maintainability, reliability, and efficiency, and the Jovial language designers attempted to achieve these goals by providing the programmer with tools for modular system decomposition, sharing of data among modules, powerful and flexible control constructs, and simple input-output schemes. The technique of sharing via a COMmunications POOL, for example had evolved prior to the design of Jovial. Similarly, the use of complex tables of data on small machines of the 1950's led to techniques of packing data that had to be supported by any high-level language for these systems. Finally, since these systems were written by sophisticated programmers, a rich concise notation was preferred to the loose English-like style of Cobol.

Ironman: The other three languages were initially defined in the second half of the 1950's; Ironman, on the other hand, will be defined twenty years later. Thus, both the specific application domain and a great deal of cultural context differs. The notions of program structuring, potentially dangerous language constructs, and software engineering did not explicitly exist before the mid 1960's. Thus Ironman is responding to a range of concerns quite different from those facing the designers of the other languages. These responses can be seen in the restrictions on the *goto*, the restrictions on aliasing (see section 3.2.1.3), and in the explicit inclusion of an encapsulation mechanism.

3.1.2 Selection vs. Synthesis

As noted earlier, the philosophical question here is whether to attempt to provide one of every possible feature or to provide a basic set from which more complex things can be created. It is interesting that the languages are schizophrenic on this question; in particular they often seem to respond differently to data and control issues.

Fortran: Fortran takes the menu approach to data; it provides a small collection of scalar data types and N-dimensional arrays of these scalar types. With respect to control, on the other hand, Fortran takes a synthetic approach. The only "high-level" control constructs are the DO and subroutine call; all others must be synthesized from the IF and GOTO.

Cobol: Cobol's data definition mechanism is essentially synthetic. It differs from the modern view of type (or structure) composition, however, in that it really only permits the description of a hierarchical naming of a linear sequence of characters. The Cobol attitude toward operations is strongly menu-oriented; the language makes a serious attempt to provide a comprehensive set of verbs and all possible options suitable for each verb. At the same time, the language prevents user-extension of these options. Cobol's attitude toward control is, again, synthetic, with heavy use of the PERFORM verb as the basis of the synthesis. Note, however, that the complex options on many of the other verbs subsume effects that would appear as explicit control statements in other languages.

Jovial: Jovial tends to be more synthetic than either Fortran or Cobol, but with restrictions. The TABLE concept, for example, is essentially an array of records (in contemporary terminology), but the records are restricted to contain only scalars (ITEMS). Thus data structures are synthesized, but the level of nesting is limited. Note that although this limitation may be aesthetically unsatisfying, is probably not a serious practical concern. Most interesting data structures probably fall within these limitations. The Jovial approach to control is primarily synthetic, but with a menu of primitives strongly influenced by modern notions of what constitutes a good set of primitives. (Note that the version of Jovial we are considering was defined in 1974.)

Ironman: Ironman, in most respects, favors the synthetic approach. The encapsulation mechanism is the strongest example of this, particularly since the report requires that types introduced by the programmer have an equal status with those predefined by the language. With respect to control, however, Ironman tends to have more of a menu than the other languages being compared -- presumably this is because the issues of "well-structured" control constructs have been so thoroughly discussed in the past several years. Nevertheless, this larger set is still intended as a basis for programming by synthesis.

3.1.3 Degree of Permissiveness

As noted above, this question is concerned with the degree to which a language attempts to prevent a programmer from powerful, but dangerous, programming practices. The primary places in which this attitude is reflected are in:

1. The ability to create aliases.[10]

2. The ability to circumvent the language's type-checking mechanism.

[10] An "alias" is an alternative naming path to access a given variable. When aliases, or "synonyms" exist, it is possible to modify the variable in non-obvious ways.

3. The degree to which the language permits or demands run-time checks (e.g., of array subscripts).

4. The degree to which the language explicitly exposes the run-time representation, for example the representation of arrays or the exit-value of a loop index.

Fortran, Cobol, and Jovial all tend to be permissive (albeit in somewhat different ways) while Ironman, defined in a very different cultural context, tends to be more restrictive. COMMON and EQUIVALENCE in Fortran both permit explicit aliasing, for example. Although these features were originally intended only to effectively utilize the scarce primary-memory resource, they are extensively used to circumvent the type mechanism. Similarly, the based tables in Jovial and REDEFINES in Cobol reflect a generally permissive attitude with respect to accessing memory. The restrictive flavor of Ironman is exhibited by the requirement for strong type-checking and the intention to prevent aliasing, although there are questions as to whether the present language requirements actually do so.

The permissive attitude of the earlier languages can be traced largely to two sources: (1) a less jaded attitude about what could be accomplished in software, and (2) the fact that the smaller machines of that era required more careful use just in order to get the job done. The second of these concerns is still evident in Ironman since some of the applications will run on very small minicomputers or microcomputers; however, Ironman avoids the most obvious pitfalls by demanding that permissive constructs be used only in encapsulations.

3.2. Programming in the Small

This set of comparisons deals only with language features of the cores and with the impact of these features on the construction of individual procedures or small programs. The features addressed here affect the ease with which algorithms and data structures can be described, they affect the programmer's ability to construct "well structured programs", and thus they affect the legibility of the resulting code. They do not necessarily seriously impact the organization of large program systems (although there are some features which impact both).

3.2.1 Syntax Issues

The grammar rules of a language affect the programmer's ability to read and write the language fluently. It is important for a programmer to be able to think about the problem and algorithm; he should not waste energy worrying about how to express the algorithm. Thus, the uniformity, expressiveness, clarity, and size of the language are of major significance. If the language definition is large, as those for Fortran and Cobol are, a programmer may not be able to know the entire language and, as a result, may code in a private sublanguage. Abstraction mechanisms can also help readability in particular by focussing the text on appropriate details.

3.2.1.1 Regularity of Grammar

The grammars of the languages under comparison have different degrees of regularity -- that is, they reuse definitions of substructure to a greater or lesser extent.

Fortran: The syntax of Fortran is "flat", with a relatively small list of features and common rules for forming substatements. These rules are not, however, uniformly applied; special restrictions apply, for example, to the expressions that may appear as subscripts ([Fortran 66] 5.1.3.2). (Interestingly, most of these restrictions are relaxed by most compilers even though they still appear in the standard.)

Cobol: Cobol is defined in terms of a set of individual features; each feature is tailored to its own task and has a syntax appropriate to that task. Thus each verb has its own grammar, and the rules are not uniform across features. There are, however, some common concepts (such as "sentence") that crop up periodically.

Jovial: The Jovial language is explicitly defined in terms of a recursive grammar. This leads to a higher degree of uniformity. The format of the ITEM declaration, for example, is also used to describe the fields of table entries. Since some options of an ITEM declaration, however, are not appropriate to a field description, the grammar is slightly different.

Ironman: The requirements are quite explicit about syntactic uniformity: for example, it must be possible to write an expression anywhere both a variable and a constant of the same type may appear. However, there are exceptions; for example, the "short-circuit" evaluation of boolean expressions in conditionals and iterations is not necessarily identical to the full evaluation of boolean expressions which is required elsewhere.

We have used the word "regularity" where others have used "involution", "orthogonality", and "symmetry". All of these words are intended to connote the desirable property that the programmer need only learn a small number of concepts that may then be used in any of a large number of contexts. It seems clear that Fortran, Jovial, and Ironman form a spectrum of increasingly regular syntax, and that within this spectrum, greater regularity is better. Cobol, however, is interesting in that it rejected the idea that uniformity is an inherent "good", and chose instead to adopt a specialized, "natural" syntax for each of its constructs.

3.2.1.2 Readability and Intelligibility

A language is ***readable*** to the extent that the form and style of programs in the language simplify the mechanical aspects of reading. A program is intelligible to the extent that what is read can be understood. Generally a program must be readable in order to be intelligible; the proper choice of comments, mnemonic identifiers, consistent indentation, and so on all contribute to the understandability of a progrm. These factors are under the control of the programmer, not the language, and are usually more important than language-dependent factors. Nevertheless, with this caveat we can make a few remarks about the readability of the various languages.

Fortran: The "flatness", and especially the statement-per-line format of Fortran programs, permits but does not encourage indentation or other devices to aid the visual impact of the program text. Numeric statement labels prevent mnemonic labeling of parts of a program.

The comment convention in Fortran (a line with a "C" in column 1) prevents comments from appearing on the same line as program text; this reduces the density of information and may interfere with clarity. Both COMMON and EQUIVALENCE introduce the possibility of using different names to refer to the same item of data. This may significantly decrease the intelligibility of the text because a simple (readable) statement could have unexpected effects unless the programmer is aware of possible aliases. The implicit type coercions may decrease intelligibility because each implementation may choose a different order for applying them.

Cobol: The designers of Cobol had a farsighted grasp of the great importance of the readability of program text. Often, however, the pseudo-English prose style of Cobol hinders this goal as much as it helps it. Individual statements are easy to read, but the verbosity of this style substantially (and often unnecessarily) increases program size -- thus decreasing intelligibility. Both the use of 88-levels as predicates (or as enumerated types) and the data definition facility emphasize the distinction between data and control, thus removing unnecessary knowledge about the data representation from the algorithms which deal with it. (However, a programmer may need a cross-reference listing to find the definition sites of data items or 88-level predicates.) The implicit editing operations may cause an innocent-looking data transfer to have unexpected side effects. Although the programmer must be concerned with certain margins for placement of statements, within those margins the language is free-form, and techniques such as indentation may be used to increase the readability and intelligibility of the program text.

Jovial: Of the three existing languages being compared, Jovial comes closest to the generally accepted notions of readable languages -- in particular its free-form input, unobtrusive comment convention and a tradition of indented coding all contribute to readability. Several aspects of Jovial, however, tend to detract from its readability. Some of these are minor; for example, the fact that the separator between the boolean expression and "then-part" of an IF is a semicolon, rather than *then* or some other mnemonically significant delimiter, may be confusing. Similarly, single letter symbols are used (in declarations) to denote the type of the variable being declared; this both prohibits the use of single character identifiers and may make the declarations cryptic. The use of DEFINE, a simple macro definition facility, permits a programmer to have significant positive or negative effects on both readability and intelligibility depending upon the taste with which it is used. Properly used, macros can suppress irrelevant detail and thus substantially improve readability; misused they may obscure important information and make a program illegible.

Ironman: Readability is, of course, an explicit goal of the Ironman requirements. The degree to which this goal will be achieved is not known. However, the designers have a great deal more information at their disposal than did the designers of the three existing languages. We expect that the goal can be achieved.

All of the languages except Fortran provide a mechanism for (logically) including information from a library in the compilation process (COPY in Cobol, COMPOOL in Jovial, and "libraries" in Ironman). All of these mechanisms create the opportunity to hide irrelevant detail and thus increase the readability of the resulting program. But such mechanisms also create the opportunity to tuck important information in obscure places -- thus decreasing intelligibility. This, like so many other language features, cannot be judged in isolation; its desirability depends upon intelligent use.

We recommend that the interested reader examine Weissman's study [Weissman73] of the effect of various factors that are commonly thought to affect the intelligibility of programs, e.g., indentation, mnemonic identifiers, etc. The overall impression given by this study is that intelligibility is affected less by language features than one might expect. Thus, again, we emphasize that the above discussion must be tempered by the realities of its actual import.

3.2.1.3 Synonyms or Aliasing

The ability to generate synonyms, or aliases, will impact large program organization as well as local readability. Thus the following remarks also apply to the concerns of section 3.3.1. If there are two or more naming paths by which the same storage location can be addressed simultaneously, we say that synonyms, or aliases, exist for that location. Most aliasing arises in one of the following ways: (1) a global variable is passed as a parameter to a subroutine, so the subroutine can name the variable as both the formal parameter and as the global itself, (2) the same variable name is passed to a subroutine in two parameter positions, so the subroutine can name it with either formal name, (3) the language supports a "reference", or "pointer" data type, and two references to the same variable can be created, and (4) an explicit "remapping" operation (or declaration) can appear in the language (e.g., EQUIVALENCE in Fortran).

In some cases, the ability to create synonyms can increase the readability of a program. Often, however, it can lead to subtle, but profound, interactions and hence to obscure programs. The position of the four languages with respect to aliasing are:

Fortran: The primary sources of aliasing in Fortran arise in connection with COMMON and EQUIVALENCE, but the problem of passing the same actual parameter in two argument position also exists. Although the language standard appears to prohibit passing a variable in COMMON as a parameter to a subroutine that modifies either the parameter or its location in the COMMON region, we do not believe this can be checked by a compiler. The ability, indeed the need, to use separate COMMON declarations in each subroutine encourages aliasing -- although (safe) conventional practice is to use identical declarations in each subroutine. The EQUIVALENCE declaration was initially intended for overlays, but it has often been used to create aliases which subvert the typing mechanism.

Cobol: Although Cobol is largely immune to some forms of aliasing because it has no parameter-passing mechanism, the REDEFINES clause in a data description and the multiple-records feature for an input file allow a programmer to explicitly set up an alias by declaring different identifiers to represent the same storage. The Cobol programmer is encouraged to use REDEFINES as a means of building and processing mixed-record files, but he must be careful not to forget that two records are aliased.

Jovial: Jovial has all the aliasing problems mentioned above: parameters can reference global variables, two non-scalar formal parameters may reference the same actual, and based tables provide both the effect of pointer-variables and arbitrary remapping.

Ironman: The Ironman requirements specifically prohibit features that permit the creation of aliases. It is not clear, however, that the report is internally consistent on this issue; elsewhere in the report, for example, Algol-like scope rules are required, and such rules lead to aliasing of type (1) discussed above. Thus, the intent appears to be to prohibit aliasing, but the specifics of how this will be done are unclear.

It seems clear that completely unconstrained aliasing is incompatible with modern software engineering practice, and it certainly precludes any possibility of program verification. The practicalities of the situation are unclear, however. No language that prohibits all aliasing has yet been used to construct a significant piece of software.

3.2.1.4 Abstraction Facilities

The readability of even small sections of program text is affected by the kind and amount of detail exhibited in the text. An abstraction mechanism is one which permits the programmer to define all the details of the implementation of a conceptual entity in one, localized place. The most familiar abstraction mechanism is, of course, the subroutine. Subroutines, however, provide only operational (ie., algorithmic) abstractions; modern programs also requires data control abstractions.

Many of the topics discussed here affect large program organizations (section 3.3.1.1), but they are also of significance in the present context. All the languages provide procedures for control abstraction, but each provides some other facilities:

Fortran: The SUBROUTINE is the only abstraction provided in Fortran; there are no means for defining abstract data structures or control structures. In some cases a stereotyped use of COMMON and EQUIVALENCE may allow the programmer to mimic non-homogeneous structures (records), but only at some risk of error.

Cobol: The only subroutine mechanism provided in standard Cobol is a set of PERFORMed paragraphs; this mechanism is quite weak, however, since it does not admit parameterization. The definition of data structures and 88-level definitions are powerful abstraction tools since they remove specific formats and values from the algorithmic portion of a program. in particular, the implicit editing associated with a MOVE suppresses an enormous amount of (irrelevant) detail.

Jovial: In addition to a more sophisticated procedure mechanism (particularly with respect to parameter binding) and data structuring, Jovial provides a macro-definition facility called DEFINE. The DEFINE can be used in the most obvious way to construct in-line procedures; in addition, however, it can be used to construct data and control abstractions. As with all macro mechanisms, it is possible to misuse the facility and create obscure programs. By encapsulating DEFINEs in COMPOOLs, however, it is possible to achieve much the same effect as with modern encapsulation mechanisms such as that proposed for Ironman.

Ironman: The report requires an explicit encapuslation mechanism to support the definition of abstractions; it further requires that types defined in this way have equal status with the built-in types of the language. The report, on the other hand, does not require a macro facility; this may represent a reaction to the undisciplined use of macros in earlier languages. It remains to be seen whether the encapsulation facility can replace all the legitimate use of macros in earlier languages; in particular, we suspect that the lack of a facility for defining control abstractions may be a serious omission which, at this stage of technology, would be best supplied through a macro mechanism.

3.2.2 Data Issues

Since the primary purpose of most programs is to transform data from one form to another, the data types and organizations provided by the language have a profound effect on the resulting programs. The important aspects of a language's data structuring facilities include its view of data types, the mechanisms for defining nonscalar structures, the degree of user control over data organization and representation, and the interaction between types and operations.

3.2.2.1 Language View of Type

The fundamental view of typing in a language emerges through the set of primitive types provided and the degree of enforcement of type rules (that is, the "strength" of type checking). We first compare the primitive types provided by the core languages.

Fortran: Real, integer, logical.

Cobol: Decimal and character strings with formatting, binary (computational-1), and various special usages (display, computational-2)

Jovial: Character string, signed and unsigned integers, floating point, and bits.

Ironman: At least real, integer, fixed point, and logical.

The extent of the influence these type selections have on the programmer is impacted by the degree to which type associations are enforced. That is, languages that check type correspondences under all circumstances force the programmer to be more attentive to his data than do those in which the type declaration means little more than a comment. The impact of type checking is also affected by the richness of the underlying type structure; if this is lean, information that might be carried in the types may have to be encoded in data. Enumerated types, for example, are superior to explicit integer encoding, even when they result in identical object code.

The strength of type checking is intimately related to the amount of coercion (i.e., automatic type conversion) permitted by the language. Coercions occur when the data to be used in some context has a type other than that explicitly anticipated by the program text. Performing the conversion automatically can serve as a convenience to the programmer, but it can also produce unexpected effects without warning. Type checking can also be defeated through aliasing;

since aliasing problems are discussed in section 3.2.1.3 we shall not repeat the discussion here.

Fortran: Typing is weak, and can easily be circumvented. There is no checking of the types of parameters across subroutine calls; both COMMON and EQUIVALENCE explicitly permit aliasing of dissimilar types to the same storage locations. The coercions between operands of type REAL and type INTEGER are implicit and may lead to unexpected results.

Cobol: The notion of type is a little fuzzy; the primary orientation is clearly to a single scalar type (character representations in any of several formats), but COMPUTATIONAL provides for various binary representations. The notion of coercion is correspondingly fuzzy, but certainly more central to the Cobol philosophy than the corresponding concept in other languages. This is especially evident in the implicit editing associated with data movement; a great deal of work is accomplished by simply moving a field in one format into one in another format.

Jovial: The Jovial type mechanism is both richer (i.e., has more primitive types) and stronger than that of Fortran. The consistency of actual and formal parameters is checked on all subroutine (procedure) calls, including separately compiled procedures linked through a COMPOOL. Coercions are defined, however, and the BASED facility can be used to subvert the type checking mechanism. Also, although the STATUS declaration provides much of the power of enumerated types, checking is not provided. Finally, there is no explicit *type* definition or encapsulation facility but these facilities can be simulated through intelligent and disciplined use of DEFINE, STATUS, and COMPOOL.

Ironman: The requirements demand strong typing with a rich set of primitive types and structuring tools (arrays, records, references). The programmer may also define new types which acquire the same status as the primitive types. Coercions between types are prohibited. Given all this, one presumes that the authors intend strict typing, but there are exceptions. The report also calls for: (1) assignment between any two records with corresponding structure even though they may not be the same conceptual type, and (2) an implicit assignment for programmer-defined types even though it may not be sensible for the conceptual type.

The sequence <Fortran, Jovial, Ironman> provides a series of increasingly powerful type mechanisms. As in other things, Cobol is not directly comparable.

3.2.2.2 Nonscalar Data Organizations

In the previous section we addressed the primitive scalar data types provided by the language. These languages also provide ways to compose these into array or record structures which give the languages a major part of their flavor.

Fortran: The only aggregate structure is the rectangular homogeneous array of scalar elements. Neither records nor pointers are permitted, so all non-array structures must be modeled in terms of arrays. Moreover, because there is no macro facility, the full detail of the representation encoding must appear explicitly at each use site.

Cobol: The language provides arrays and records. The record structure may be deeply nested, and parallel equivalencing of corresponding substructre is available. There are no pointers (references), however.

Jovial: Scalars may be composed into records, arrays, and arrays of records (called TABLES). There are no explicit references (pointers), but unsigned integers may be used as addresses; this is an unsafe, but very powerful, substitute.

Ironman: The requirements call for homogeneous arrays, records, and pointers, and for these to be recursively definable -- thus, one may have arrays of records which, in turn, contain records, and so on. The requirements also specify recursively-definable structures which, if achieved, will eliminate the need for pointers wherever explicit sharing is not required.

3.2.2.3 Definition and Representation of Data

A user may exercise control over data definitions at two levels: at the language level by defining new abstractions and at the *machine level* by controlling the low-level representation on the hardware.

At the high level, we must further distinguish between data organizations provided by the language itself and facilities provided to the programmer for application-specific structures. These might be as simple as enumerated types or as complex as extensions to the basic type structure of the language.

Fortran: The programmer has no direct control over representation; the only representation decision (that of arrays) is preempted by the language. The fact that the representation of arrays is explicitly defined in the language permits the clever programmer to encode other representations, but only at the price of having the mapping function appear at each use.

Cobol: Although the record structure of Cobol is predominantly oriented to processing character streams, it allows a quite explicit and very rich choice of representations. The 88-level definitions may be used as both enumerated types and simple predicates.

Jovial: Jovial provides extensive control over representation, most notably including BASED records for explicit address arithmetic. Status lists provide a simple form of enumerated types.

Ironman: As noted earlier, Ironman requires a complete facility for programmer definition and use of encapsulated types. The ability to embed machine code in an encapsulation presumably provides arbitrarily fine-grained representation control.

Access to the underlying physical representation of data is also important. It is of major concern to programmers who must interface with given constraints, either in time, space, or preexisting data representations.

Fortran: The physical layout of arrays is part of the language specification. Although generally not considered an advantage, knowledge of the representation of arrays both permits some coding efficiencies and compensates for Fortran's otherwise arid data structuring.

Cobol: Secondary storage is an integral part of the programmer's world view. Its format can be explicitly encoded in the data division. File and data formats, including exact field sizes, can be defined. Thus, the programmer may explicitly specify much of the representation to be used by the underlying hardware. Note that, although Cobol is generally regarded as strongly machine-independent, this kind of low-level control can cause intolerable inefficiencies on some machines.

Jovial: Precision of numeric items can be specified in declarations, and the locations of fields within a TABLE entry can be specified down to the bit level in some forms of the TABLE declaration.

Ironman: The language shall permit, but not require, programs to specify the physical representation of data, which seems to imply that the compiler is free to choose a representation unless specifically instructed to use some particular one.

3.2.2.4 Interaction Between Data and Operations

The contemporary view of data types holds that types are characterized by both a set of values and a set of operations on those values. Thus the kinds of operations a language provides on its primitive data and the consistency of those operations from one type to another strongly affect the tone of the language. We must consider what kinds of operations are available, whether they can be applied to aggregate structures or only to scalars, and the degree to which they are implicitly invoked.

Fortran: The only operations are the scalar operations on the primitive types. The only implicit operations are the integer-real coercions. The only means of extending the set of operations is through SUBROUTINES.

Cobol: Rich record-oriented operations are provided, and the effects of such verbs as MOVE are strongly controlled by the data structures and field types involved. Operators are composed from scalar operations in a fashion appropriate to the type and structure of the data. There is no purely algorithmic abstraction mechanism, however.

Jovial: The only operations are scalar operations on the primitive types, and there is no means (other than PROCEDURES) for extending these operations. Character strings, however, are among these primitive types. Nonscalar TABLE structures can be remapped to other TABLE structures by means of the BASED TABLE mechanism.

Ironman: In addition to all the usual scalar operations and procedure mechanism, the report requires that it be possible to augment the built-in operators (e.g. plus) to operate on new user-defined types.

3.2.3 Control Issues

We will consider separately the commands used to describe the flow of control *within a subprogram and the facilities for inter-module control.*

3.2.3.1 Local Control

The considerations of modern programming methodology have identified a number of control constructs whose use leads to more manageable, understandable programs. These constructs are :

1. A grouping syntax (for making compound statements or blocks).

2. Two flavors of alternatives, an *if* and a case (either an enumerated or an indexed multi-way alternative construct).

3. Two styles of loops, a ***while*** loop that iterates until a specified condition is satisfied, and a ***for*** loop which sequences through a specified set of values and assigning each value in the sequence to a "control variable".

We will not repeat the arguments that support the selection of these as the central modern control commands. The four languages under consideration provide the following constructs:

Fortran: Only the simple one-way *if* and integer counting *for* are provided. There are no compound statements or *while* loops.[11]

Cobol: Paragraphs and sections serve to construct compounds. The IF allows grouping of statements at a level finer than the paragraph and provides a two-way alternative construct. The loop construct, based on the PERFORM verb, allows iteration to be controlled by either or both of condition-testing and indexing through a range.

Jovial: Algol-style compounds, *if*, and *for* are provided as well as a *while*.

Ironman: All of the modern "well-structured" control constructs are required by the report.

Despite the theoretical soundness of the arguments in favor of the "well-structured" command set, certain practical situations (notably processing (local) exceptions) are awkward to handle with those commands alone. Therefore, in addition to the "well-structured" control set, all these languages provide an explicit *goto*. However, the particular flavor of any *goto* is colored by the possible destinations of the jump as determined, for example, by the scope of the statement labels.

Fortran: Except for certain semantic restrictions concerning inactive DO loops which are difficult to enforce at compile time, a GOTO may branch to any statement within the current program unit.

Jovial: Scope rules make it impossible to jump into an inner compound statement or loop body except when the destination label is explicitly exported from its block. Jumps into inactive procedures have undefined results.

[11] A multi-way branch exists in the full language but has been excluded from the core because of its restricted power; it can be composed from a sequence of IF...GOTO constructs and in itself provides no more power or flexibility than this. The impact of the multi-way alternative is more profound in languages which permit nested synthetic structures.

Cobol: Any paragraph may be the destination of a *goto*, whether or not it is part of an active loop. The language definition insures that all transfers will work correctly and not interfere with any existing loop state.

Ironman: The *goto* is required, but an explicit list of restrictions constrains it to the uses generally recognized as appropriate in situations where well-structured control is awkward.

3.2.3.2 Nonlocal Control

In addition to the local control commands described above, all the languages provide inter-module control in the form of subroutine calls. Although these constructs are primarily concerned with organizing the relations among groups of programs, they also impact local programming. The inter-module issues are discussed in section 3.3; here we focus on the local issue.

The subroutine is the primary tool for defining (and isolating) abstractions in most traditional languages; as such we have come to rely heavily on its properties. The properties that are important local programming are: (1) how the body is delimited, (2) parameter conventions and options, (3) and local variable conventions and options. Some of these topics are covered elsewhere in this report, but we shall repeat them here briefly for reference.

The bodies of subroutines in the various languages are delimited as follows:

Fortran: The last statement is (must be) an END.

Cobol: A subroutine is not syntactically delimited at its definition. Rather, at the call site the set of paragraphs comprising the body are specified in the call (i.e. the PERFORM). Note that this implies that the same paragraph may be executed in line or as part of several different (conceptual) subroutines.

Jovial: The body of a PROCEDURE is a statement. Note, however, that a statement can be either a compound or a block, so the body may in fact be arbitrarily complex.

Ironman: The requirements do not specify the syntax for a procedure body, but we may presume that, like Jovial, it will be a statement.

The parameter conventions are:

Fortran: The implementor may use either call-by-value-result or call-by-reference.

Cobol: There are no parameters to PERFORMed paragraphs.

Jovial: Call-by-value, call-by-result, and call-by-value result are provided for scalars; TABLES are passed by reference only.

Ironman: At least call-by-value, result, and value-result are required.

The facilities for local variables include:

Fortran: The scope of a local variable is the program unit in which it is declared. Its extent is that of the program unit's activation.

Cobol: There are no (syntactially) local variables.

Jovial: Both dynamically allocated (IN) and statically allocated (RESERVE) locals are available.

Ironman: As in Jovial, both dynamically and statically allocated locals are available.

It is clear that, of the existing languages, Jovial has the strongest subroutine mechanism. The Cobol mechanism is so weak as to be almost non-existant.

3.2.4 Efficiency Concerns

Despite the fact that the need for "efficiency at all costs" has decreased as hardware has become less expensive and faster, there are some ways, and some applications, in which the efficiency of a language significantly affects the way in which it is used. In particular, programmers will contort a program's organization in order to avoid constructs that are known to be inefficient;

typically, for example, procedures (subroutines) are larger in PL/I programs because programmers are aware of the overhead associated with calling them. Similarly, programmers will warp the organization of a program in order to use those features that are known to be efficient -- as in the extensive use of COMMON in Fortran programs.

The impact of efficiency concerns arises in both programming "in the small" and "in the large"; we have placed the discussion here because it was the first opportunity to do so. The reader should be aware that these remarks actually apply in both contexts.

The first paragraph of this section used the phrase "the efficiency of a language"; this phrase requires some explanation. The major factor in determining the efficiency of a particular program is generally the proper choice of data structure and algorithm for that program -- factors outside the influence of the language. Usually the next most important factor is the quality of the particular implementation of the language -- a factor beyond the control of the language designer. In addition, however, the language design may prevent highly efficient implementation. It is these factors that are of concern in a language comparison, and even then they are of concern only to the extent that they are likely to impact good program organization.

Designed in the late fifties, when efficiency was a more pressing concern, both Fortran and Jovial have known efficient implementations; both contain features (e.g., COMMON and COMPOOL) and restrictions (e.g., on recursive subroutines) that permit efficiencies. Both also contain some features that interfere with optimization techniques that were discovered since the time the languages were defined. For example, the extensive use of GOTO in Fortran complicates global flow analysis and prevents some optimizations that might be applied if the compiler knew which control construct were being synthesized. Similarly, the possibility of a non-local GOTO in Jovial requires some extraneous run-time overhead. However, these points are probably second-order effects.

The dominance of input-output in most Cobol applications tends to decrease the emphasis on extremely efficient implementations; however, there is little in the language that prevents such implementations. The feature-oriented nature of the language, for example, permits the ambitious compiler-writer to apply a great deal of special case knowledge to optimizing the most frequently used constructs. In addition, Cobol affords the programmer considerable control over the representation of his data (e.g., via USAGE).

Since Ironman is not yet a language, it is difficult to comment on its efficiency. The requirements do call for a number of constructs that are generally understood to be inefficient, but it is not clear that these will survive to the language design stage. Moreover, the philosphy espoused by the report recognizes the need for efficiency in the intended application domain, and it explicitly warns against the inclusion of inefficient constructs.

3.3. Programming in the Large

The languages we are concerned with in this report are used to write (the components of) large programming systems. A language comparison must therefore address the support they provide for problems that arise when programs are thousands of lines long, when programs are in use for many years, or when several programmers are involved in the design, implementation, and maintenance of a program system.

There are a number of software engineering methodologies for dealing with large programs. They differ in details, but they share a common view: When a project involves long times and several people, it must be possible to:

1. Decompose the task into subtasks which can be worked on independently.

2. Assemble the resultant pieces into a coherent, operational whole.

3. Manage the requisite long-term maintenance and enhancement.

Support for the tasks of decomposition, assembly, and maintenance must be provided in many ways, not just through the programming language. For example, when more than a single person is involved in these activities a number of management and coordination problems arise. A tool, e.g. a language, cannot solve all of these problems, but it may facilitate their solution. Thus, in this section we shall consider those aspects of the languages under consideration that impact these issues.

3.3.1 Decomposition of the System

A system may be decomposed along natural lines in any of several dimensions. Two obvious dimensions are functionally and data-oriented, but there may be others and, of course, a single system may be decomposed along more than one dimension simultaneously.

We will discuss each of these dimensions presently; whatever the nature of the decomposition, however, the resulting pieces must correspond to "work assignments" [Parnas72]. That is, the decomposition is not useful unless it permits a portion of the original task to be performed by a person (or team) in relative isolation. As we shall see, the major impact of language issues on "programming in the large" is the degree to which a language permits or encourages such independence.

By a *functional* decomposition we mean one which emphasizes major algorithmic components of the system. Often such components are temporally related: First one component is applied to the data, then the next, and so on until the task is complete. This is the traditional view of program decomposition and is typified by elaborate and detailed specifications of the data structures which form the "interface" between the functional subunits.

By a *data-oriented* decomposition we mean one which emphasizes the major (abstract) data structures and the operations defined on them. This view of decomposition is more modern and, although not proven on a large number of systems, is gaining a great deal of favor. It is typified by careful, often mathematical, specifications of the abstract data structures in terms of their invariant properties and the operations which may be applied to them.

Since no practical decomposition is purely functional or purely data-oriented, the issues related to both become entwined in the following discussion.

Languages can affect the ability to decompose a system into modules in two important ways. We must first ask what kinds of modular decompositions are supported by the language; that is, we must examine the facilities for defining and using abstractions. We must then consider the extent to which they ensure the actual independence of separate work assignments.

3.3.1.1 Definition and Use of Abstractions

Abstraction is one of the most important tools of the modern software engineer. The language designer's choices about abstraction facilities have a strong impact on the availability of functional or data-oriented abstraction. This, in turn, determines which system designs (i.e., decompositions) are practically feasable.

Almost all languages provide some methods for separating the definitions of isolable concepts from the use of those concepts: at a bare minimum, procedures will be available. We see in the core languages a range from simple procedures to complex facilities for extending the type structure of the base language.

Fortran: The sole isolable concept in Fortran is the abstract operation described by a sub-program. There is no way to isolate the definitions of data in a common location such that all program units can share one definition. This is particularly noticeable in the need to include the type and DIMENSION specifications (as well as EQUIVALENCE declarations) independently in every program unit. The language specifies no way to "encapsulate" abstractions (e.g., in the form of macros) that represent arbitrary syntactic forms. Thus, many abstractions in data and control that are not captured in either COMMON or sub-program declarations must be written out explicitly in terms of the primitive data and operations at each usage.

Cobol: The physical representation of data can be defined in a single definition, which can be copied into a source program with the COPY declaration. The operations on the data can be specified in computation units of paragraphs or sections, which can also be shared among many modules in a large system. However, because of the need to pass parameters to such operations through global storage, the mechanism is weaker than that of Fortran. The level-88 predicates allow a weak form of enumerated type to be shared across modules. Other abstractions that cannot be captured by the data definitions, level-88 predicates, or in paragraph bodies must be explicitly defined in terms of the primitive data and operations at each usage.

Jovial: The physical representation of data can be defined in a COMPOOL file by using based tables. The abstract operations can be defined by including macros or procedure declarations in the COMPOOL. The definitions of enumerated types (STATUS variables) can be shared in the same way. Other "encapsulated" abstractions, such as macros that expand into arbitrary syntactic forms in the language, are communicated through COMPOOL in the DEFINE declaration.

Ironman: Modern ideas about abstraction have significatly influenced the Ironman requirements. This is evident in both the general design criteria and specific requirements for rich data structuring, programmer-defined types, generic definitions, strong library support, and encapsulation.

3.3.1.2 Enforcement of Independence

When a system is decomposed into isolated tasks that are created as independent work assignments, it is important that each programmer know *how* other program segments will depend on his. The most important term of support is clear specification of the interface, but that is (at present) not a language issue. *Some language facilities do support this independence, however.* The degree to which abstractions can be encapsulated (section 3.3.1.1) is important, as is the ability to avoid aliasing (section 3.2.1.3). A third set of features affect the programmer's ability to control when and by whom his data may be accessed. These later features include scope and extent rules, allocation policies, and declarations.

The scope, extent, and allocation rules for a variable determine the lifetime of the variable and the portions of the program in which it can be accessed. Scope and extent rules for a variable are usually, but not always, determined by its declaration. Allocation rules are generally inflexible; at most a small menu of allocation policies are available.

Fortran: Data names are local to a program unit. COMMON labels and subroutine names, and only those, are global to all program units. *Because of the flat syntax, there is no intermediate scope between* these. Allocation is static, so the extent of a variable is the lifetime of the program.[12] The EQUIVALENCE facility allows the user to specify that logically separate data areas may physically overlap. Implicit variable declarations allow new variables to be created inconspicuously, and sometimes erroneously.

Cobol: All names are global to a program module, although some names (e.g., field names of structures) may be ambiguous if incompletely qualified. A single level of hierarchy is available for procedure labels by the use of SECTION declarations; all paragraph names are local to a section. The language specification explicitly states how the compiler shall disambiguate data and paragraph names that are not fully qualified. Allocation is static, although there are facilities (e.g., REDEFINES) that allow data areas to overlap.

Jovial: Most variable names obey the rules of simple nested block structure. The COMPOOL facility provides nonlocal scope through mutual agreement of the affected modules. Allocation for nested blocks can be handled with a runtime stack, and the BASED construct allows an (unsafe) escape through which the programmer may define a dynamic storage allocation mechanism.

Ironman: It is clear that block-structured scope rules are intended, but the report is confusing about some details (see section 2.4.3). Some clauses require simple nested block structure, some require scope to be lexically determined, and others require encapsulations to provide protection and private data. Moreover, aliasing is prohibited. It is unclear that these requirements can be satisfied simultaneously. A runtime stack will suffice for all allocation except for dynamic types -- but it is not clear how complex the stack mechanism must be. Dynamic types will require a heap with garbage collection.

[12] Note, however, that the standards permit an implementation to perform dynamic allocation of named COMMON; see [Fortran 76] sections 8.3.5 and 15.8.4 and [Fortran 66] section 10.2.5.

3.3.2 Assembly of a System from Components

The task of assembling a number of independent modules into a coherent, operational system is usually regarded as a problem in integration and testing. As such, it is supported primarily by nonlinguistic tools such as file librarians, linkers, debugging systems, and analysis packages. Although (as for decomposition) the language is not the dominant factor affecting the task, some properties of the language may be of substantial help or hindrance. These aspects of the language are concerned with the ways independently-developed units can be coupled and with the safety of the assembly process.

In section 3.3.1 we discussed the ways a system can be separated into independent subproblems. In that discussion we were concerned with the ***logical*** separation of modules. However, if individuals or small teams are to work independently, it is extremely helpful to be able to separate the tasks ***physically*** as well. The degree to which the language supports the development of physically independent modules impacts the integration process and the testing style. The amount of support the language provides for making sure that interface assumptions are observed is also important.

3.3.2.1 Physical Independence of Modules

The most common way to support physically independent module development is to provide a means for independently compiling the various modules of a system. This provides a boundary for protection of data; it may facilitate independent testing; and it may allow substantial savings in compilation time.

In order to establish a common context between separately-compiled modules, particularly in a functional decomposition, it is necessary to communicate both the structure and the location of the data that is being shared. This context may be established by linking separate code segments into a common address space, or it may be loosely coupled, with external data serving to carry the common information. In sharing, it is at least necessary to communicate the location of the code, and perhaps other properties as well (e.g., number and/or types of the parameters).

Fortran: The structure of shared data is communicated through COMMON declarations. The location of the data is communicated through the link-edit process. The location of code is established by the link-edit process as well, but in general there is no attempt to assure that parameters correspond in type and number. Facilities for this checking are not part of the Fortran language specification.

Cobol: The sharing of data is strongly affected by the view that secondary storage is an integral part of a program's name space. Data definitions provide the structure of the data; the COPY verb simplifies maintenance of this information by allowing many programs to extract data definitions from a common library. The location of data in secondary storage is established by the environment division or by the operating system command language. The flavor of the data-sharing is dominated by the language's strongly serial world view. Communication of data is usually done by sharing secondary storage between separate programs rather than by sharing primary storage between modules. This limits the grain of interaction of program segments. There is no facility like explicit COMMON that permits separately compiled modules to share data in primary memory. Code may be shared by using the COPY verb to include source code from a common library.

Jovial: COMPOOL provides many of the same facilities as Fortran COMMON and Cobol COPY. In particular, the type of shared variables and the type and number of formal parameters of shared procedures is shared.

Ironman: There are requirements for libraries, sharing, and link-editing of information. Encapsulated type definitions allow information about data and associated operations to be communicated implicitly through the language's type structure.

3.3.2.2 Checking Module Linkages

The only form of module specification provided in current languages takes the form of type and number checking of external declarations and parameters. External linkages were discussed in the previous section; we turn now to parameters.

Parameter binding rules determine whether the actual parameter may be used for input or for output, whether it is reevaluated each time it is used, and whether or not its value is dynamically updated during procedure execution. In addition, a language may or may not enforce type checking across procedure calls, particularaly when separately compiled modules are involved.

Fortran: The language specification is (intentionally) vague on whether scalar parameters are called by reference or by value-result. The distinction affects whether certain aliasing problems arise. The specification is quite explicit on how arrays shall be passed as parameters (by reference) and furthermore states explicitly how these are to be interpreted. In any case, the language is defined in such a way that the type and number of the parameters passed to an external module cannot be checked for validity. Furthermore, a number of restrictions in the Fortran language standard (which arise because of aliasing) cannot be checked by the compilers (e.g., [Fortran 76] section 15.9.3.5 or [Fortran 66] section 8.4.2). Thus, large systems require careful documentation to assure that the language restrictions are not inadvertently violated. The language standard conspicuously does not specify what will happen if the restrictions are violated.

Cobol: The concept of externally-defined procedures is not part of the language[13] . However, the PERFORM verb allows a paragraph or group of paragraphs to be invoked as a subroutine. Since the data in the program is global to such a subroutine, a paragraph can potentially access (and affect) all of it. Parameters to such subroutines are necessarily communicated through global storage.

Jovial: A syntactic and semantic distinction is made between input and output (scalar) parameters; a parameter used both for input and output must be mentioned twice. Tables are always passed by reference. Type checking is firm, but can be circumvented.

[13] Some common extensions to COBOL, which are neither part of the core nor part of the language standard, permit separately compiled modules to communicate via a subroutine CALL mechanism with explicit parameter passing.

Ironman: Type checking is strong, but the definition of what constitutes a type (with respect to parameters) is unclear. In any case, it appears to be the intent that aliasing be prohibited. Input and output parameters are distinguished. Input parameters act as constants within a procedure invocation.

3.3.3 Maintenance and Enhancement

We have grouped together maintenance (repairing errors) and enhancement (adding features or improving performance) because they involve similar activites and are both done on "operational" systems. Experience, often bitter experience, suggests that these activities are both costly and error-prone, and hence deserve special attention in the design of the tools (e.g., languages) used.

The most outstanding characteristic needed in a maintainable system is ***understandability***. The person responsible for maintaining a system is usually not one of its original authors -- and even when it is, the interactions in large system are usually subtle, complex, and easily forgotten. Thus the first problem of maintenance is to (re)understand the existing code well enough to design and implement the necessary changes.

Some aspects of understandability are discussed in the sections on programming in the small, 3.2. However large programs introduce additional problems. It is no longer the case that the definition and use of each variable (or subroutine) will appear within a few pages of each other. One cannot simply "flip pages" to find how a variable was declared, whether it is ever used in particular ways, or whether certain properties of it are assumed by the code. Indeed the definition and uses of a variable (or subroutine) may not even be in the same module (or file), and responsibility for maintenance of the modules may reside with different people. Thus, while the ability to indent, to use nicely-structured control constructs, and so on, are helpful for understanding local parts of the system, they do not address some of the essential properties of large-system maintenance.

The previous paragraph emphasizes one aspect of maintaining large systems -- the need to be able to locate relevant information. The dual problem, which is even more important, is to hide irrelevant information. Humans are not well suited to coping with vast amounts of detail, and systems maintainers are no exception. The maintainer needs to be able to function with an abstract model of the components of the system -- to be able to understand the components in terms of what they do rather than the details of how they do it. When errors are to be fixed or changes made he uses this model to guide him to the code to be modified -- and, equally importantly, to suggest what code can be ignored as irrelevant to the current task.

There is a clear tension between these two aspects of making a large system understandable; some information must be easy to find, other information must be hidden. Modern methodlogy, of course, holds that systems can be structured, often hierarchically, so that at a given level there is a clear separation between what must be known and what must be hidden. We subscribe to that view but it must be remembered that: (1) it often takes an especially perceptive individual to devise a suitable structure for a given system, and (2) none of the existing languages was designed at a time when these issues were as clear as they now are. Thus, only Ironman, through its requirement for libraries and encapsulations, provides direct support for these modern ideas.

The notions of libraries and encapsulations as required in the Ironman report are close to the most modern notions of what is needed to support large program maintenance; thus they are the yardstick against which we shall measure the facilities of the existing languages. Since the relevant language features have been mentioned previously we shall only summarize them here.

Fortran: As noted before, the only abstraction tools are the subroutine and named COMMON. Since there is neither type checking of parameters nor consistency checking of COMMON declarations, there is no language support for enforcing understandability-in-the-large. Only careful conventions and documentation make large system construction feasible.

Cobol: Very large systems are constructed in Cobol, yet there is virtually no language support for understandability-in-the-large.

Jovial: The COMPOOL, except for rigorous type-checking, comes close to the Ironman facility.

Chapter 4
Conclusions

The purpose of this report was the comparison of four programming languages to determine their appropriateness for use with current software engineering methodologies. In chapter 1 we set forth a new technique for language comparisons. This methodology involves identifying a core that captures the characteristic properties of each of the languages, then couching the comparisons in terms of these cores. Chapter 2 defines a core for each of the four languages; chapter 3 discusses the way these core languages respond to various methodological issues.

We shall now review this approach and summarize the important points in the cores and the comparisons.

4.1. On Describing Languages Through Cores

We developed the core approach to language comparison for much the same reason that motivates us to devise abstractions for our programs. Full language definitions contain an enormous amount of detail; most of it, however, is not significant for our goals. By concentrating on the essence of the languages, we are able to focus on the properties that in fact influence the way programs are written. It is these properties that should be compared to satisfy our goals, and the suppression of other information eliminates many distractions from the discussion. The resulting analysis is, we believe, more pertinent to the goal than a comprehensive comparison of details would have been.

The cores for the three existing languages are all short -- the syntax definitions are about one page long -- but they span the actual languages well. Indeed, we had to deviate from the core languages only slightly to write runnable versions of the example programs, and those deviations were primarily concessions to our operating system. Throughout the comparisons, we found that the cores included the aspects of the languages that our intuitions argued are pertinent to software engineering.

This is not to say that any subset of a language which is complete, or almost complete, can serve as a core for a study such as this. On the contrary, defining a representative core requires substantial sensitivity to language issues in general and to the languages under comparison in particular. It is easy, for example, to define a computationally complete subset of Fortran that fails to capture the flavor and power of the language. Once a representative core has been defined, however, it sharpens the issues and leads to more concrete comparison.[14]

4.2. Remarks on the Comparisons

The dimensions along which we compared the core languages were chosen to emphasize our view of software engineering. We see three major interactions between language design and software engineering practice. First, the language designer's implicit assumptions about programs and programming have subtle but important effects on the language and its usefulness as a tool. Second, the language is the primary tool for actually writing the algorithms that control computations. As such, its uniformity and expressive power has a strong impact on the programmer at those times when he is actually generating code. This is programming in the small. Third, all significant systems are made up of many components and the choice of language affects both the decomposition of a system into components and the styles of communication among those components. These considerations impact multi-person projects -- that is, programming in the large.

4.3. Remarks on Languages

This report emphasizes comparison, not evaluation. A few evaluative remarks are in order, however.

[14] It is interesting to consider whether the core approach could serve as a training tool or an organizational principle for a course in language comparison. We are enamoured of the idea, but it is inappropriate to pursue it here.

None of the three existing languages is ideal for modern methodology. We feel strongly, therefore, that there is much to be gained by a well-thought-out language design that meets (or exceeds) the Ironman requirements.

In the light of modern software engineering practice, we feel that the issues of programming in the large are central. The COMPOOL mechanism of Jovial is the most useful tool provided by any of the three existing languages; it allows sharing of definitions, procedures, and data, and this serves as a weak encapsulation mechanism. The Cobol COPY mechanism shares many of these virtues.

There seem to be no dimensions along which Fortran dominates the other languages. Thus, although a strong Fortran implementation and good system support may provide a pragmatic advantage over weaker implementations of the other languages, Fortran seems to have no linguistic advantage.

Finally, there is much still at stake in the Ironman proposal. The proposed requirements generally reflect what has been learned about programming languages and software engineering over the past twenty years. Certain ambiguities and contradictions pointed out in this paper, however, indicate the remaining danger that the gains of Ironman could fall well short of what is possible.

Acknowledgements

We are particularly grateful to Carolyn Councill and Howard Wactlar for system support provided during the preparation of this report, and to Roy Weil (of Michael Baker, Jr., Inc.) for his insightful comments on the use of programming languages in practical contexts.

References

[AirForce 76] Department of the Air Force, ***Military Standard Jovial (J3)***, MIL-STD-1588 (USAF), Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York 13441, June 1976.

[Cobol 60] CODASYL, ***Initial Specifications for a COmmon Business Oriented Language***, Department of Defense, April 1960.

[Cobol 74] X3.23-1974 ***American National Standard COBOL***, 1974

[DEC 76] Digital Equipment Corporation, ***DECsystem-10 COBOL: Programmer's Reference Manual***, Digital Equipment Corporation publication DEC-10-LPCRA-B-D, Maynard, Massachussetts, 1976.

[Fortran 66] X3.9-1966 ***American National Standard Fortran***, American Standards Association, Inc., New York, N. Y., 1966

[Fortran 76] X3.9-1976 ***Draft Proposed ANS Fortran***, published in SIGPLAN Notices, 11, 3 (March 1976).

[Goodenough 76] Goodenough, J. B., ***A Study of High Level Language Features: Detailed Language Feature Analysis***, SOFTECH, Inc., Waltham, Ma., February 1976.

[Higman 67] Higman, B., ***A Comparative Study of Programming Languages***, American Elsevier, New York, 1967.

[IBM 57] International Business Machines Corp., ***General Information Manual: Programmer's Primer for FORTRAN Automatic Coding System for the IBM 704 Data Processing System***, Form No. 32-0306-1, 1957.

[IBM 61a] International Business Machines Corp, *General Information Manual: FORTRAN*, IBM F28-8074-1, December 1961.

[IBM 61b] International Business Machines Corp., *Reference Manual, 709/7090 FORTRAN Programming System*, IBM C28-6054-2, January 1961.

[Ironman 77] Department of Defense, High-Order Language Working Group, *Department of Defense Requirements for High-Order Computer Programming Languages: "Ironman"*, Department of Defense, Jan. 1977.

[Nicholls 75] Nicholls, J. E., *The Structured Design of Programming Languages*, Addison-Wesley, 1975.

[Parnas 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15, 12, December 1972 (pp. 1053-1058).

[RADC 76] Rome Air Development Center, *Jovial J73/I Specifications*, Rome Air Development Center, Air Force Systems Command, Griffis Air Force Base, New York 13441, July 1976.

[Ralston 71] Ralston, A., *Introduction to Programming and Computer Science*, McGraw-Hill, 1971.

[Sammet 69] Sammet, J. E., *Programming Languages: History and Fundamentals*, Prentice-Hall, Englewood, 1969.

[Shaw 63a] Shaw, C. J., "Jovial--A Programming Language for Real-Time Command Systems", *Annual Review of Automatic Programming*, 3 (1963).

[Shaw 63b] Shaw, C. J., "A Specification of JOVIAL", *Communications of the ACM* 6, 12 (Dec 63).

[Strawman 75] Report of Subcommittee on Strawman HOL Requirements.

[Tinman 76] Department of Defense Requirements for High Order Computer Programming Languages: "Tinman", April, 1976.

[Weissman 73] "Psychological Complexity of Computer Programs: An Initial Experiment", ***Computer Systems Research Group Technical Report***, University of Toronto, 1973.

[Woodenman 75] ***"Woodenman" Set of Criteria and Needed Characteristics for a Common DOD High Order Programming Language***, Institute for Defense Analysis, August 1975.

Appendix A
Some Anomalies in the Ironman Requirements

The discussions of ambiguities and inconsistencies in section 2.4.3 require detailed documentation. In this appendix we paraphrase and summarize the points in the Ironman report [Ironman 77] that support the arguments in the text. Note that we have done this only for the anomalies. The principles discussed in sections 2.4.2.1 to 2.4.2.4 are supported consistently by the report, and a sampling of the requirements should suffice.

A.1. Scope Rules

The dominant philosophy calls for nested, block-structured scope rules, but many requirements conflict with that position.

1A: The language is to be general only to the extent necessary to satisfy the requirements.

1B: Error-prone features are to be avoided.

1C: Readability and maintainability are to be emphasized.

1D: The language shall aid production of efficient code and avoid distributed overhead.

1E: The language shall be small and simple.

1F: The language shall be composed from features that are understood and implementable.

3C: The scope of a type definition shall be determinable at translation time.

3-3J: Elements of dynamically allocated types may be created at will and must remain allocated as long as they are accessible.

3-5B: Encapsulation inhibits exportation of variable and operation names of the implementation.

3-5C: Encapsulations may declare private variables; the scope of these variables is the same as the scope of the encapsulation. It is not clear whether such variables are to be defined once for each encapsulation or once for each use of an encapsulation as a type.

3-5D: The scope rule for encapsulations must be violable.

4C: Few side effects in expressions are permitted.

5C: The intended scope of a declaration shall be determinable at compile time. Scopes may be lexically embedded.

5C: Warnings shall be issued when a name is redefined, but functions with different specifications are not the same.

6A: Local scopes shall be allowed in control statements.

6G: Some ideas about scope are implicit in the restrictions on the destinations of gotos.

7C: Name inheritance is block-structured and static (lexical).

7E: Function calls may not alter either their parameters or (other) variables in the caller's scope.

7I: Procedure calls are restricted so that two parameters or an inherited name and a parameter cannot be aliases for each other, either directly or indirectly.

8E: It shall be possible to override the normal control over logical and physical resources. This presumably includes allocation policies that support block structure.

10C: The invocation of an exception causes a transfer of control whose destination is unclear.

11E: The safety of encapsulated machine code shall be maximized, but machine code shall be permitted.

12B: Separately compiled segments shall explicitly export and import shared definitions.

A.2. Types

A.2.1 Strong Type Checking

Strong typing is clearly intended, but the following statements appear in the requirements:

1B: Error-prone features are to be avoided.

1F: All language restrictions must be enforceable by the translator.

3A: The type of each expression or component of an expression shall be determinable at compile time.

3B: Implicit type conversions are forbidden.

3C: New type definitions shall be allowed; they shall be processed entirely at translation time.

3-1C: Explicit conversions between floating point precisions shall not be required; in other words, the precision of a floating point number does not affect its type.

3-1F: Similarly, the range of an integer or fixed point variable does not affect its type.

3-1H: However, explicit scale conversions are required as necessary, so the scale of a fixed point variable does affect its type.

3.2: It is not clear whether or not two enumerated types with the same elements are of the same type.

3-2B: A variable of an enumerated type may be restricted to a subrange. It is not clear whether distinct subranges are distinct types.

3.3: Similarly, it is not clear whether or not two composite types with the same structure are of the same type.

3-3B: Range, precision, and scale are all mentioned in the discussion of component type specification.

3-3D: The number of dimensions of an array shall be fixed at compile time, but the range of array subscript values may be determined at allocation time. In view of 3A, the exact range must not affect the type of the array.

3-3F: Assignment shall be permitted between records with corresponding components of identical name and type. Since assignment cannot coerce types, such records must be of the same type. It is not clear whether this requirement applies if a subset of the components correspond. The situation is further muddled by 3-3G, which allows individual record components to be read-only.

3-3H: Variant types are allowed if the variant is resolvable at compile time. It is not clear how the value of a variable can be converted from one variant to another.

3-3I: Dynamically allocated types with dynamic substructure are allowed. They must be distinguishable from other composite types. It is clear that variables with dynamic substructure can violate strong typing.

3-5A: Type definitions may be encapsulated. It is not clear when, if ever, two encapsulations define the same type.

3-5D: Operations such as type conversion may operate simultaneously on more than one type. It is not clear whether the definitions of such operations will appear to violate strong typing.

4B: It shall be possible to determine the type of each expression at compile time.

4D: Expressions are allowed wherever variables and constants of the same type are allowed.

5B: Variables may be of any type.

5D: Procedures, functions, types, labels, exceptions, and statements are not types.

7G: Range, precision, and scale specifications are required for formal parameters as part of the type-checking provisions.

7H: The number of dimensions for array parameters must be determinable at compile time, but the exact ranges may vary from call to call.

8C: Input shall be allowed only from files whose representation is known at compile time.

11B: More than one physical representation may be used for variables of a single type.

11E: Embedded machine code is allowed; it can clearly violate typing.

12B: Type checking shall be enforced across the boundaries of separately compiled segments.

12D: Generic type definitions are allowed.

A.2.2 Programmer-Defined Types

A second set of unclear requirements concern the use of programmer-defined types.

1E: The language shall have uniform syntactic conventions.

1F: All language restrictions shall be enforceable by the compiler.

2.2: Some literal values are built in.

3C: New type definitions are permitted. No restriction shall be imposed on defined types unless it is imposed on all types.

3-2A: Literals of enumeration types shall be syntactically distinguishable from other identifiers. It is not clear what this means. Also, equality is automatically defined.

3-3A: It is not clear whether the composition rules for composite types are limited to arrays and records. It is not clear whether recursive definitions are allowed.

3-5A: Encapsulated type definitions shall be allowed.

4E: Constant-valued expressions shall be evaluated before execution time. It is not clear how to do this for programmer-defined types.

5F: Assignment and value access operators shall be automatically defined for each variable. This implies that programmer-defined types must have those operators.

7A: Existing operators may be extended to new data types.

A.3. Time/Space Tradeoffs

The language has ambitious goals for programmer control of time/space tradeoffs, but it is not clear they are achievable.

1D: Users shall be able to specify the time space tradeoffs in a program. Constructs with unusually cheap or expensive implementations should be easily recognized.

1F: It should be possible to predict the interactions of features.

3-3G: Nonassignable record components need not take data storage space.

3.3.3: Dynamically allocated types are almost forced to undergo the overhead of garbage collection.

7B: Recursive procedures shall be permitted, but they may not increase the execution costs of other constructs.

8E: It shall be possible to override the language's resource-management policies.

11A: Programs may specify the physical representation of data.

11B: More than one representation at a time may be used for a single type.

11E: Embedded machine code shall be allowed.

11F: It shall be possible in programs to specify the optimization criteria to be used. This shall include the ability to express preferences for optimizing translation time, execution time, or runtime storage requirements.

12D: The compilation of generic definitons may share object code.

13A: The defining documentation might point out the relative efficiency of alternative constructs. (But note 1E: the language should not provide several notations for the same concept.)

13D: Translators shall warn of the use of expensive constructs.

A.4. Hardware Access vs. Machine Independence

Ironman strives to provide both access to the underlying hardware and independence of specific implementation details.

1A: Applications require real time control, self diagnostics, input-output to nonstandard devices, and file processing.

1C: Defaults shall be severely restricted.

1D: Features should be implementable efficiently for many object machines.

1G: The language shall strive for machine independence, but there shall be a facility for specifying those portions of a program that are dependent on the object machine configuration.

2A: All constructs shall be representable in the 64-character ASCII subset.

3-1C: Floating point precision specifications shall be interpreted as minimum precision, and rounding or truncation shall use the implemented precision.

3-1D: Floating point computations may use object machine hardware properties. Built-in operations shall make actual properties accessible.

3-1E,3-1H: Integer and fixed-point values and operations shall be treated exactly as specified.

3-3I: Dynamic types are intended primarily for the support portions of embedded computer software.

8A: Low-level input/output operations shall act directly on physical files, channels, and devices.

8D: The language shall not require an operating system, and standard definitions shall be independent of one if it exists.

8E: A few low-level facilities shall expose physical resources and allow language-controlled management policies to be preempted.

9D: Real time constraints shall be expressible and checkable.

9E: There shall be a real time clock.

AD-A053 562 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
A COMPARISON OF PROGRAMMING LANGUAGES FOR SOFTWARE ENGINEERING.(U)
APR 78 M SHAW, G T ALMES, J M NEWCOMER F30602-75-C-0218

UNCLASSIFIED RADC-TR-78-58 NL

2 OF 2
AD
A053 562

END
DATE
FILMED
6-78
DDC

10A: Exceptions to be processed include hardware-dependent conditions.

11A: Programs may specify the physical representation of data.

11C: The language shall require global constants to specify the object machine configuration.

11D: Programs may use machine-dependent facilities, but such usage shall be permitted within conditional structures that discriminate on the machine facilities.

11E: Escape to machine code shall in some cases be possible, but it shall be encapsulated and implemented as safely as possible.

13A: It should be possible to predict program behavior from the language definition.

13E: Machine independent parts of translators should be separate from code generators.

13F: Restrictions imposed by translators should reflect real limitations of object machines, not arbitrary decisions for convenience.